Algorithms for Optimization

Algorithms for Optimization

SECOND EDITION

Mykel J. Kochenderfer
Tim A. Wheeler

The MIT Press
Cambridge, Massachusetts
London, England

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.



© 2026 Massachusetts Institute of Technology

Copyright in this monograph has been licensed exclusively to The MIT Press, http://mitpress.mit.edu. All inquiries regarding rights should be addressed to The MIT Press, Rights and Permissions Department.

This book was set in TEX Gyre Pagella by the authors in LATEX.

To our families.

Contents

Preface xv

Acknowledgments xix

1 Introduction 1

1.1 A History 2
1.2 Optimization Process 4
1.3 Mathematical Formulation 5
1.4 Applications 7
1.5 Minima 10
1.6 Optimality Conditions 10
1.7 Overview 14
1.8 Summary 20
1.9 Exercises 20

2 Derivatives and Gradients 23

2.1 Derivatives 23
2.2 Derivatives in Multiple Dimensions 25
2.3 Numerical Differentiation 26
2.4 Automatic Differentiation 29
2.5 Regression Gradient 35
2.6 Simultaneous Perturbation Stochastic Gradient Approximation 37
2.7 Summary 38
2.8 Exercises 39

3 *Bracketing* 43
3.1 Unimodality 43
3.2 Finding an Initial Bracket 43
3.3 Fibonacci Search 45
3.4 Golden Section Search 47
3.5 Quadratic Fit Search 51
3.6 Shubert-Piyavskii Method 53
3.7 Bisection Method 57
3.8 Summary 59
3.9 Exercises 59
4 *Local Descent* 61
4.1 Descent Direction Iteration 61
4.2 Step Factors 62
4.3 Line Search 64
4.4 Approximate Line Search 65
4.5 Trust Region Methods 71
4.6 Termination Conditions 73
4.7 Summary 74
4.8 Exercises 74
5 *First-Order Methods* 77
5.1 Gradient Descent 77
5.2 Conjugate Gradient 79
5.3 Momentum 81
5.4 Nesterov Momentum 82
5.5 AdaGrad 82
5.6 RMSProp 84
5.7 Adadelta 84
5.8 Adam 86
5.9 Hypergradient Descent 86
5.10 Summary 89
5.11 Exercises 89

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

6 Second-Order Methods 95
- 6.1 Newton's Method 95
- 6.2 Secant Method 97
- 6.3 Levenberg-Marquardt Algorithm 100
- 6.4 Levenberg-Marquardt for Sum of Squares 102
- 6.5 Quasi-Newton Methods 103
- 6.6 Summary 106
- 6.7 Exercises 106

7 Direct Methods 113
- 7.1 Cyclic Coordinate Search 113
- 7.2 Powell's Method 115
- 7.3 Hooke-Jeeves 116
- 7.4 Generalized Pattern Search 117
- 7.5 Nelder-Mead Simplex Method 119
- 7.6 Divided Rectangles 124
- 7.7 Summary 131
- 7.8 Exercises 131

8 Stochastic Methods 135
- 8.1 Noisy Descent 135
- 8.2 Mesh Adaptive Direct Search 136
- 8.3 Memory-Efficient Zeroth-Order Optimization 138
- 8.4 Simulated Annealing 140
- 8.5 Cross-Entropy Method 144
- 8.6 Proposal Distribution Descent 148
- 8.7 Information-Geometric Optimization 149
- 8.8 Covariance Matrix Adaptation 152
- 8.9 Summary 156
- 8.10 Exercises 156

9 Population Methods 163
- 9.1 Population Iteration 163
- 9.2 Genetic Algorithms 164
- 9.3 Differential Evolution 172
- 9.4 Particle Swarm Optimization 172

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

9.5 Firefly Algorithm 174
9.6 Cuckoo Search 177
9.7 Hybrid Methods 179
9.8 Summary 179
9.9 Exercises 181

10 Constraints 183

10.1 Constrained Optimization 183
10.2 Constraint Types 184
10.3 Transformations to Remove Constraints 185
10.4 Removing Affine Equality Constraints 187
10.5 Projected Iterative Methods 188
10.6 Lagrange Multipliers 189
10.7 Inequality Constraints 192
10.8 Slack Variables 195
10.9 Penalty Methods 195
10.10 Method of Multipliers 197
10.11 Interior Point Methods 199
10.12 Summary 201
10.13 Exercises 201

11 Duality 209

11.1 Dual Problem 209
11.2 Primal-Dual Methods 213
11.3 Dual Ascent 216
11.4 Alternating Direction Method of Multipliers 219
11.5 ADMM Applications 223
11.6 Distributed Methods 234
11.7 Summary 239
11.8 Exercises 239

12 Linear Programming 249

12.1 Problem Formulation 249
12.2 Simplex Algorithm 255
12.3 Dual Certificates 266
12.4 Summary 269
12.5 Exercises 269

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

13 *Quadratic Programming* 273
- 13.1 Problem Formulation 273
- 13.2 Unconstrained Least-Squares Problems 274
- 13.3 Least Squares with Linear Inequalities 278
- 13.4 Least Distance Programming 281
- 13.5 Solving Nonnegative Least-Squares Problems 285
- 13.6 Dual Certificates 287
- 13.7 Summary 290
- 13.8 Exercises 291

14 *Disciplined Convex Programming* 297
- 14.1 Canonical Form 297
- 14.2 Verification 298
- 14.3 Canonicalization 307
- 14.4 Solving 314
- 14.5 Summary 317
- 14.6 Exercises 317

15 *Multiobjective Optimization* 325
- 15.1 Pareto Optimality 325
- 15.2 Constraint Methods 329
- 15.3 Weight Methods 331
- 15.4 Multiobjective Population Methods 334
- 15.5 Preference Elicitation 340
- 15.6 Summary 344
- 15.7 Exercises 344

16 *Sampling Plans* 349
- 16.1 Full Factorial 349
- 16.2 Random Sampling 350
- 16.3 Uniform Projection Plans 351
- 16.4 Stratified Sampling 352
- 16.5 Space-Filling Metrics 352
- 16.6 Space-Filling Subsets 356
- 16.7 Quasi-Random Sequences 359
- 16.8 Summary 361
- 16.9 Exercises 363

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

17 *Surrogate Models* 365
17.1 Fitting Surrogate Models 365
17.2 Linear Models 366
17.3 Basis Functions 367
17.4 Fitting Noisy Objective Functions 372
17.5 Model Selection 375
17.6 Multifidelity Surrogate Models 383
17.7 Summary 384
17.8 Exercises 385
18 *Probabilistic Surrogate Models* 389
18.1 Gaussian Distribution 389
18.2 Gaussian Processes 391
18.3 Prediction 394
18.4 Gradient Measurements 396
18.5 Noisy Measurements 399
18.6 Fitting Gaussian Processes 401
18.7 Summary 401
18.8 Exercises 402
19 *Surrogate Optimization* 411
19.1 Prediction-Based Exploration 411
19.2 Error-Based Exploration 412
19.3 Lower Confidence Bound Exploration 413
19.4 Probability of Improvement Exploration 413
19.5 Expected Improvement Exploration 414
19.6 Safe Optimization 416
19.7 Summary 424
19.8 Exercises 424
20 *Optimization Under Uncertainty* 427
20.1 Uncertainty 427
20.2 Set-Based Uncertainty 429
20.3 Probabilistic Uncertainty 432
20.4 Summary 439
20.5 Exercises 439

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

21 *Uncertainty Propagation* 443
21.1 Sampling Methods 443
21.2 Taylor Approximation 444
21.3 Polynomial Chaos 445
21.4 Bayesian Monte Carlo 455
21.5 Summary 457
21.6 Exercises 457
22 *Discrete Optimization* 463
22.1 Integer Programs 464
22.2 Rounding 465
22.3 Cutting Planes 466
22.4 Branch and Bound 471
22.5 Dynamic Programming 474
22.6 Ant Colony Optimization 477
22.7 Summary 481
22.8 Exercises 481
23 *Expression Optimization* 489
23.1 Grammars 489
23.2 Genetic Programming 493
23.3 Grammatical Evolution 497
23.4 Probabilistic Grammars 501
23.5 Probabilistic Prototype Trees 502
23.6 Summary 509
23.7 Exercises 510
24 *Multidisciplinary Optimization* 515
24.1 Disciplinary Analyses 515
24.2 Interdisciplinary Compatibility 517
24.3 Architectures 521
24.4 Multidisciplinary Design Feasible 521
24.5 Sequential Optimization 524
24.6 Individual Discipline Feasible 526
24.7 Collaborative Optimization 531
24.8 Simultaneous Analysis and Design 534
24.9 Summary 535
24.10 Exercises 536

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

appendices

A Julia 541
- A.1 Types 541
- A.2 Functions 555
- A.3 Control Flow 558
- A.4 Packages 560

B Test Functions 561
- B.1 Ackley's Function 561
- B.2 Booth's Function 561
- B.3 Branin Function 564
- B.4 Flower Function 565
- B.5 Michalewicz Function 566
- B.6 Rosenbrock's Banana Function 567
- B.7 Wheeler's Ridge 568
- B.8 Circle Function 569

C Mathematical Concepts 571
- C.1 Asymptotic Notation 571
- C.2 Taylor Expansion 573
- C.3 Convexity 574
- C.4 Norms 577
- C.5 Generalized Gradient 577
- C.6 Matrix Calculus 579
- C.7 Positive Definiteness 582
- C.8 Matrix Decompositions 582
- C.9 Gaussian Distribution 586
- C.10 Gaussian Quadrature 586

References 591

Index 607

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Preface

This book provides a broad introduction to optimization with a focus on practical algorithms for the design of engineering systems. We cover a wide variety of optimization topics, introducing the underlying mathematical problem formulations and the algorithms for solving them. Figures, examples, and exercises are provided to convey the intuition behind the various approaches.

This text is intended for advanced undergraduates and graduate students as well as professionals. The book requires some mathematical maturity and assumes prior exposure to multivariable calculus, linear algebra, and probability concepts. Some review material is provided in the appendix. Disciplines where the book would be especially useful include mathematics, statistics, computer science, aerospace, electrical engineering, and operations research. Although we will occasionally refer to particular applications as examples, the focus is on general algorithms and their properties rather than the details of specific application domains.

Fundamental to this book are the algorithms, which are all implemented in the Julia programming language. We have found the language to be ideal for specifying algorithms in human readable form. Permission is granted, free of charge, to use the code snippets associated with this book, subject to the condition that the source of the code is acknowledged. We anticipate that others may want to contribute translations of these algorithms to other programming languages. As translations become available, we will link to them from the book's webpage.

The implementations of the algorithms in this book are intended to convey the essential ideas behind each approach. These implementations are not intended to be production quality, but rather to be clear and concise. There are many optimization software packages, both commercial and open source, that provide much more robust implementations. We reference many of these packages in the

text, and we encourage readers to explore these packages for practical applications. In some situations, especially for problems that do not lend themselves to general-purpose solvers, the Julia code in this book may be useful starting points for custom implementations.

We intentionally wanted to optimize for breadth rather than depth—unifying ideas in optimization that span many different communities that have emerged from diverse disciplines ranging from operations research to aerospace design. There are many excellent optimization books published over the past fifty years that go deep into particular areas, such as convex optimization, discrete optimization, and linear programming. Instead of attempting to reproduce their theoretical analyses, we reference them in the wide margins.

New in this second edition are three chapters on duality, quadratic programming, and disciplined convex programming. The first new chapter covers the principles of duality and several of the core optimization algorithms that leverage its theory, including primal-dual methods, dual ascent, and the alternating direction method of multipliers. The previous edition had sections that mentioned duality, but dedicating a chapter to it allows for more focus on its theory and primal-dual algorithms. We are particularly excited to cover the alternating direction method of multipliers, an approach that has been the subject of recent theoretical and practical advances. This algorithm is generally applicable to constrained optimization and can be robustly scaled to extremely large problems. We apply the alternating direction method of multipliers to a broad range of problems that generalize under its umbrella, including alternating projections, least absolute deviation, and basis pursuit.

The second new chapter covers quadratic programming. It mirrors our existing chapter on linear-constrained optimization, now renamed ''Linear Programming''. Linear programming refers to solving a very general class of linear-constrained optimization problems with linear objectives. Quadratic programming is also a very general class of linear-constrained optimization problems, but with quadratic objectives. This new chapter covers a series of algorithmic problem decompositions that successively transform a general quadratic program into a form amenable to efficient solution by iterative algorithms.

The third new chapter focuses on disciplined convex programming. Disciplined convex programming allows problems to be expressed in a natural manner, but allows for algorithms to verify problem convexity and reformulate input problems into a general structure amenable to efficient solution by a wide variety of solvers.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Disciplined convex programming is a workhorse of modern convex optimization. This chapter covers the theory and algorithms to allow the reader to be better equipped to effectively use disciplined convex solvers, verify convexity, and exploit problem structure to efficiently solve large problems.

In addition to these new chapters, we have added additional methods, ranging from new ways to estimate gradients to multifidelity techniques. Based on feedback from multiple iterations of the course, we improved the clarity of the text to make the topics more accessible, expanded explanations, updated references, and streamlined the algorithms. There are now additional exercises, example applications, and figures.

Mykel J. Kochenderfer
Tim A. Wheeler
Stanford, Calif.
May 24, 2026

Ancillary material is available on the book's webpage:
`http://mitpress.mit.edu/algorithms-for-optimization`

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Acknowledgments

This textbook has grown from a course on engineering design optimization taught at Stanford. We are grateful to the students and teaching assistants who have helped shape the course over the past five years. We are also indebted to the faculty who have taught variations of the course before in our department based on lecture notes from Joaquim Martins, Juan Alonso, Ilan Kroo, Dev Rajnarayan, and Jason Hicken. Many of the topics discussed in this textbook were inspired by their lecture notes.

The authors wish to thank the many individuals who have provided valuable feedback on early drafts of our manuscript, including Atharva Aalok, Mohamed Abdelaty, Atish Agarwala, Ross Alexander, Yasmine Alonso, Piergiorgio Alotto, Nancy Ammar, Grayson Armour, Dylan Asmar, David Ata, Lily Barnett, Rishi Bedi, Logan Bell, Felix Berkenkamp, Raunak Bhattacharyya, Pierre Boca, Hans Borchers, Maxime Bouton, Stephen Boyd, Ellis Brown, Kovid Capildeo, Abhishek Cauligi, Mo Chen, Zhengyu Chen, Raphael Chinchilla, Vince Chiu, Hanyou Chu, Anthony Corso, Jonathan Cox, Nikhil Devanathan, Holly Dinkel, Katherine Driggs-Campbell, Thai Duong, Thimoté Dupuch, Hamza El-Saawy, Sofiane Ennadir, Daniel Fein, Kaijun Feng, Tamas Gal, Wenzhi Gao, Christopher Lazarus Garcia, Wouter Van Gijseghem, Michael Gobble, Robert Goedman, Jayesh Gupta, Aaron Havens, William Healy, William Ho, Richard Hsieh, Sydney Hsu, Der-Han Huang, Jeremy Huang, Zdeněk Hurák, Luke Hyman, Masha Itkina, Arec Jamgochian, Bogumił Kamiński, Walker Kehoe, Mindaugas Kepalas, Shogo Kishimoto, Veronika Korneyeva, Erez Krimsky, Petr Krysl, Tim Lappe, Jessie Lauzon, Ruilin Li, Ye Li, Iblis Lin, Sean Lin, William Liu, Edward Londner, Charles Lu, Miles Lubin, Marcus Luebke, Robert Lupton, Jacqueline Machesky, Ashe Magalhaes, Zouhair Mahboubi, Pranav Maheshwari, Yuki Matsuoka, Travis McGuire, Jeremy Morton, Robert Moss, Trương Minh Nhật, Longwu Ou, Santiago Padrón,

Ronald Pan, Jimin Park, Harsh Patel, Christian Peel, Derek Phillips, Brad Rafferty, Sidd Rao, Vale Rasmussen, Andreas Reschka, Alex Reynell, Stuart Rogers, Per Rutquist, Ryan Samuels, Orson Sandoval, Jeffrey Sarnoff, Cooper Shea, Chelsea Sidrane, Ilyass Sif, Sumeet Singh, Anshrin Srivastava, Nathan Stacey, Ethan Strijbosch, Andre Tkacenko, Alex Toews, Ava Tolentino, Pamela Toman, Olivia Tomassetti, Rachael Tompa, Zacharia Tuten, Alexandros Tzikas, Kemal Ure, Raman Vilkhu, Boris Vishnevsky, Yuri Vishnevsky, Julie Walker, Zijian Wang, Patrick Washington, Jacob West, Benjamin Whiffen, Adam Wiktor, Brian Wu, John Wu, Sofia Wyetzner, Esen Yel, Brandon Yeung, Anil Yildiz, Robert Young, Javier Yu, Andrea Zanette, Remy Zawislak, and Tarek Zougari. In addition, it has been a pleasure working with Marie Lufkin Lee, Christine Bridget Savage, and Susan Hartman from the MIT Press in preparing this manuscript for publication.

The style of this book was inspired by Edward Tufte. Among other stylistic elements, we adopted his wide margins and use of small multiples. In fact, the typesetting of this book is heavily based on the Tufte-LaTeX package by Kevin Godby, Bil Kleb, and Bill Wood. We were also inspired by the clarity of the textbooks by Donald Knuth and Stephen Boyd.

Over the past few years, we have benefited from discussions with the core Julia developers, including Jeff Bezanson, Stefan Karpinski, and Viral Shah. We have also benefited from the various open-source packages on which this textbook depends (see appendix A.4). The typesetting of the code is done with the help of pythontex, which is maintained by Geoffrey Poore. Plotting is handled by pgfplots, which is maintained by Christian Feuersänger. The book's color scheme was adapted from the Monokai theme by Jon Skinner of Sublime Text. For plots, we use the viridis colormap defined by Stéfan van der Walt and Nathaniel Smith.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

1 Introduction

Many disciplines involve optimization at their core. In physics, systems naturally tend toward their lowest energy state subject to physical laws. In business, corporations aim to maximize shareholder value. In biology, fitter organisms are more likely to survive. This book will focus on optimization from an engineering perspective, where the objective is to design a system that optimizes a set of metrics subject to constraints. The system could be a complex physical system like an aircraft, or it could be a simple structure such as a bicycle frame. The system might not even be physical; for example, we might be interested in designing a control system for an automated vehicle or a computer vision system that detects whether an image of a tumor biopsy is cancerous. We want these systems to perform as well as possible. Depending on the application, relevant metrics might include efficiency, safety, and accuracy. Constraints on the design might include cost, weight, and structural soundness.

This book is about the *algorithms*, or computational processes, for optimization. Given some representation of the system design, such as a set of numbers encoding the geometry of an airfoil, these algorithms will tell us how to search the space of possible designs with the aim of finding the best one. Depending on the application, this search may involve running physical experiments, such as wind tunnel tests, or it might involve evaluating an analytical expression or running computer simulations. We will discuss computational approaches for addressing a variety of challenges, such as searching high-dimensional spaces, handling problems where there are multiple competing objectives, and accommodating uncertainty in the metrics.

1.1 A History

We will begin our discussion of the history of algorithms for optimization[1] with the ancient Greek philosophers. Pythagoras of Samos (569–475 BCE), the developer of the Pythagorean theorem, claimed that "the principles of mathematics were the principles of all things,"[2] popularizing the idea that mathematics could model the world. Both Plato (427–347 BCE) and Aristotle (384–322 BCE) used reasoning for the purpose of societal optimization.[3] They contemplated the best style of human life, which involves the optimization of both individual lifestyle and functioning of the state. Aristotelian logic was an early formal process—an algorithm—by which deductions can be made.

Optimization of mathematical abstractions also dates back millennia. Euclid of Alexandria (325–265 BCE) solved early optimization problems in geometry, including how to find the shortest and longest lines from a point to the circumference of a circle. He also showed that a square is the rectangle with the maximum area for a fixed perimeter.[4] The Greek mathematician Zenodorus (200–140 BCE) studied Dido's problem, shown in figure 1.1. Others demonstrated that nature seems to optimize. Heron of Alexandria (10–75 CE) showed that light travels between points through the path of shortest length. Pappus of Alexandria (290–350 CE), among his many contributions to optimization, argued that the hexagon repeated in honeycomb is the optimal regular polygon for storing honey; its hexagonal structure uses the least material to create a lattice of cells over a plane.[5]

Central to the study of optimization is the use of *algebra*, which is the study of the rules for manipulating mathematical symbols. Algebra is credited to the Persian mathematician al-Khwārizmī (790–850 CE) with the treatise *Kitāb al-jabr wal-muqābala*, or *The Compendious Book on Calculation by Completion and Balancing*. Algebra had the advantage of using Hindu-Arabic numerals, including the use of zero in base notation. The word *al'jabr* is Arabic for restoration and is the source for the Western word *algebra*. The term *algorithm* comes from *algoritmi*, the Latin translation and pronunciation of al-Khwārizmī's name.

Optimization problems are often posed as a search in a space defined by a set of coordinates. Use of coordinates comes from René Descartes (1596–1650), who used two numbers to describe a point on a two-dimensional plane. His insight linked algebra, with its analytic equations, to the descriptive and visual field of geometry.[6] His work also included a method for finding the tangent to any curve whose equation is known. Tangents are useful in identifying the minima and

[1] This discussion is not meant to be comprehensive. A more detailed history is provided by X.-S. Yang, "A Brief History of Optimization," in *Engineering Optimization*. Wiley, 2010, pp. 3–10.

[2] Aristotle, *Metaphysics*, trans. by W. D. Ross. 350 BCE, Book I, Part 5.

[3] See discussion by S. Kiranyaz, T. Ince, and M. Gabbouj, *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*. Springer, 2014, Section 2.1.

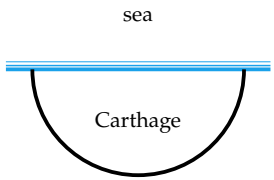


Figure 1.1. Queen Dido, founder of Carthage, was granted as much land as she could enclose with a bullhide thong. She made a semicircle with each end of the thong against the Mediterranean Sea, thus enclosing the maximum possible area. This problem is mentioned in Virgil's *Aeneid* (19 BCE).

[4] See books III and VI of Euclid, *The Elements*, trans. by D. E. Joyce. 300 BCE.

[5] T. C. Hales, "The Honeycomb Conjecture," *Discrete & Computational Geometry*, vol. 25, pp. 1–22, 2001.

[6] R. Descartes, "La Géométrie," in *Discours de la Méthode*. 1637.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

maxima of functions. Pierre de Fermat (1601–1665) began solving for where the derivative is zero to identify potentially optimal points.

The concept of *calculus*, or the study of continuous change, plays an important role in our discussion of optimization. Modern calculus stems from the developments of Gottfried Wilhelm Leibniz (1646–1716) and Sir Isaac Newton (1642–1727). Differential calculus is particularly important in guiding optimization, especially in high-dimensional spaces.[7] Joseph-Louis Lagrange (1736–1813) later unified tools from calculus and algebra to solve constrained optimization problems, where we want to optimize some objective subject to a set of constraints. He introduced the method of Lagrange multipliers, which transforms a constrained optimization problem into an unconstrained one by combining the objective function and constraints into a single function known as the Lagrangian.[8] By taking derivatives of this function and solving the resulting system of equations, he linked the continuous methods of calculus with the algebraic solution of equations, laying the groundwork for modern optimization techniques.

The mid-twentieth century saw the rise of the electronic computer, spurring interest in numerical algorithms for optimization. The ease of calculations allowed optimization to be applied to much larger problems in a variety of domains. One of the major breakthroughs came with the introduction of linear programming, which is an optimization problem with a linear objective function and linear constraints. Leonid Kantorovich (1912–1986) presented a formulation for linear programming and an algorithm to solve it.[9] It was applied to optimal resource allocation problems during World War II. George Dantzig (1914–2005) developed the simplex algorithm, which represented a significant advance in solving linear programs efficiently.[10] Richard Bellman (1920–1984) developed the notion of dynamic programming, which is a commonly used method for optimally solving complex problems by breaking them down into simpler problems.[11] Dynamic programming has been used extensively for optimal control. This textbook outlines many of the key algorithms developed for digital computers that have been used for various engineering design optimization problems.

Decades of advances in large-scale computation have resulted in innovative physical engineering designs as well as the design of artificially intelligent systems. The intelligence of these systems has been demonstrated in games such as chess, *Jeopardy!*, and Go. IBM's Deep Blue defeated the world chess champion Garry Kasparov in 1996 by optimizing moves by evaluating millions of positions. In 2011, IBM's Watson played *Jeopardy!* against former winners Brad Rutter and Ken

[7] Derivatives and gradients are reviewed in the next chapter, along with many methods for computing them. Chapters 5 and 6 show how to use derivatives and gradients to direct the optimization process.

[8] Chapter 10 introduces the Lagrangian and shows how Lagrange multipliers can be used to solve constrained optimization problems. These multipliers serve as a foundational concept for duality theory, which is covered in chapter 11.

[9] L. V. Kantorovich, "A New Method of Solving Some Classes of Extremal Problems," *Proceedings of the USSR Academy of Sciences*, vol. 28, pp. 211–214, 1940.

[10] The simplex algorithm will be covered in chapter 12.

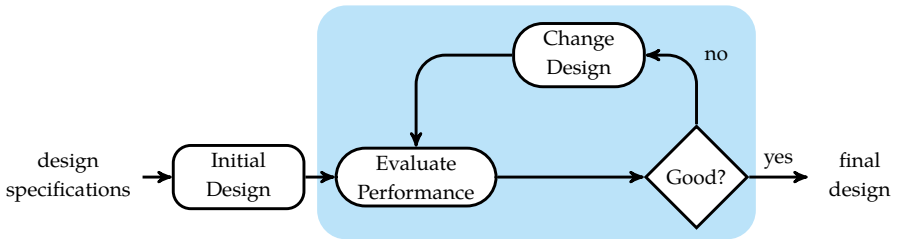
[11] Dynamic programming is reviewed in section 22.5. R. Bellman, "On the Theory of Dynamic Programming," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 38, no. 8, pp. 716–719, 1952.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Jennings. Watson won the first-place prize of $1 million by optimizing its response with respect to probabilistic inferences about 200 million pages of structured and unstructured data. Since the competition, the system has evolved to assist in healthcare decisions and weather forecasting. In 2017, Google's AlphaGo defeated Ke Jie, the number one ranked Go player in the world. The system used neural networks with millions of parameters that were optimized from self-play and data from human games. The optimization of deep neural networks is fueling a major revolution in artificial intelligence that will likely continue.[12] When ChatGPT was released to the general public in 2022, it was recognized as representing a major milestone in general-purpose artificial intelligence with its ability to generate natural language responses to text prompts, such as writing poetry or answering questions, by optimizing the parameters of a special type of deep neural network known as a transformer model with many billions of parameters.[13]

1.2 Optimization Process

A typical engineering design optimization process is shown in figure 1.2.[14] The role of the *designer* is to provide a problem *specification* that details the parameters, constants, objectives, and constraints that are to be achieved. The designer is responsible for crafting the problem and quantifying the merits of potential designs. The designer also typically supplies a baseline design or initial design point to the optimization algorithm.



[12] C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*. Springer, 2024.

[13] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, et al., "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, 2023. The transformer model was introduced by A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, et al., "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[14] Further discussion of the design process in engineering is provided by J. Arora, *Introduction to Optimum Design*, 4th ed. Academic Press, 2016.

Figure 1.2. The design optimization process. We seek to automate the optimization procedure highlighted in blue.

This book is about automating the process of refining the design to improve performance. An optimization algorithm is used to incrementally improve the design until it can no longer be improved or until the budgeted time or cost has been reached. The designer is responsible for analyzing the result of the optimiza-

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

tion process to ensure its suitability for the final application. Misspecifications in the problem, poor baseline designs, and improperly implemented or unsuitable optimization algorithms can all lead to suboptimal or dangerous designs.

There are many advantages of an optimization approach to engineering design. The optimization process provides a systematic design procedure. If properly applied, optimization can help reduce the chance of human error in design. Sometimes design intuition can be misleading; it can be much better to optimize with respect to data. Optimization can speed the process of design, especially when a procedure can be written once and then be reapplied to other problems. Traditional engineering techniques are often visualized and reasoned about by humans in two or three dimensions. Modern optimization techniques, however, can be applied to problems with millions of variables and constraints.

There are also challenges associated with using optimization for design. We are generally limited in our computational resources and time, and so our algorithms have to be selective in how they explore the design space. Fundamentally, the optimization algorithms are limited by the designer's ability to specify the problem. In some cases, the optimization algorithm may exploit modeling errors or provide a solution that does not adequately solve the intended problem. When an algorithm results in an apparently optimal design that is counterintuitive, it can be difficult to interpret. Another limitation is that many optimization algorithms are not guaranteed to produce optimal designs.

1.3 Mathematical Formulation

An optimization problem can be formulated mathematically as follows:[15]

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x} \in \mathcal{X} \end{aligned} \tag{1.1}$$

Here, $\mathbf{x}$ is a *design point*, which can be represented as a vector of values corresponding to different *design variables*. An n-dimensional design point is written $[x_1, \ldots, x_n]$, where the ith design variable is denoted x_i.[16] The elements in this vector can be adjusted to minimize the *objective function* f. Any value of $\mathbf{x}$ in the *feasible set* $\mathcal{X}$ that minimizes the objective function is called a *solution* or *minimizer*. A particular solution is written $\mathbf{x}^*$. Figure 1.3 shows a one-dimensional optimization problem.

[15] We can convert maximization problems into minimization problems by simply negating the objective function. We will default to the convention of minimizing the objective in this textbook.

[16] As with the Julia programming language, square brackets with comma-separated entries are used to represent column vectors. Design points are column vectors.

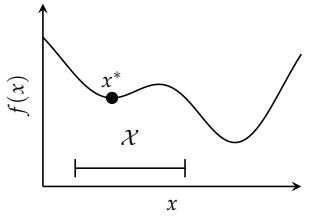


Figure 1.3. A one-dimensional optimization problem. Note that the minimum is merely the best in the feasible set—lower points may exist outside the feasible region.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The feasible set $\mathcal{X}$ may be defined in terms of a set of constraints. Each *constraint* limits the set of possible solutions, and together the constraints define the feasible set $\mathcal{X}$. Feasible design points do not violate any constraints. For example, consider the following optimization problem:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & f(x_1, x_2) \\ \text{subject to} \quad & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_1 + x_2 \leq 1 \end{aligned} \tag{1.2}$$

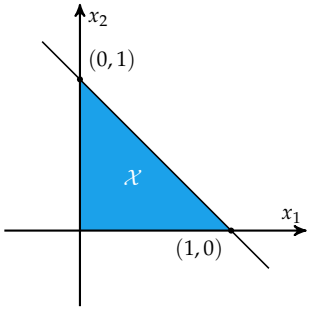


Figure 1.4. The feasible set $\mathcal{X}$ associated with equation (1.2).

In this problem, the feasible set $\mathcal{X} = \{\mathbf{x} \mid x_1 \geq 0, x_2 \geq 0, x_1 + x_2 \leq 1\}$. This set is plotted in figure 1.4.

Constraints are typically written as equalities or inequalities. If constraints involve strict inequalities, then the feasible set might not include the constraint boundary. A potential issue with not including the boundary is illustrated by

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x \\ \text{subject to} \quad & x > 1 \end{aligned} \tag{1.3}$$

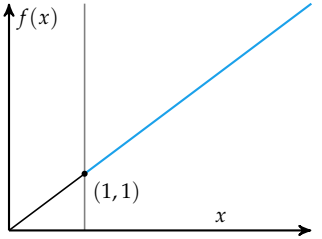


Figure 1.5. The problem in equation (1.3) has no solution because the constraint boundary is not feasible.

The feasible set is shown in figure 1.5. The point $x = 1$ produces values smaller than any x greater than 1, but $x = 1$ is not feasible. We can pick any x arbitrarily close to, but greater than, 1, but no matter what we pick, we can always find an infinite number of values even closer to 1. We must conclude that the problem has no solution. To avoid such issues, it is often best to include the constraint boundary in the feasible set.

Modeling engineering problems within this mathematical formulation can be challenging. The way in which we formulate an optimization problem can make the solution process either easy or hard.[17] In some cases, a high-fidelity representation of a problem will permit only approximate solutions. These approximate solutions may be acceptable for our particular application. Alternatively, we may want to pursue a lower-fidelity representation that permits exact solutions. We will focus on the algorithmic aspects of optimization that arise after the problem has been properly formulated.[18]

Since this book discusses a wide variety of different optimization algorithms, one may wonder which algorithm is best. There are many ways to measure the performance of an optimization algorithm, ranging from theoretical asymptotic

[17] See the preface of S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[18] Many texts provide examples of how to translate real-world problems into optimization problems. See, for example, the following: R. K. Arora, *Optimization: Algorithms and Applications*. Chapman and Hall/CRC, 2015. A. D. Belegundu and T. R. Chandrupatla, *Optimization Concepts and Applications in Engineering*, 2nd ed. Cambridge University Press, 2011. A. Keane and P. Nair, *Computational Approaches for Aerospace Design*. Wiley, 2005. P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design*. Cambridge University Press, 2017.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

complexity to empirical wall-clock time spent executing on a particular computer architecture. As elaborated by the *no free lunch theorems*, there is no reason to prefer one algorithm over another unless we make assumptions about the distribution over the space of possible objective functions. Although one algorithm may perform better than another algorithm on one distribution of problems, it will perform worse on another distribution of problems.[19] For many optimization algorithms to work effectively, there needs to be some regularity in the objective function, such as Lipschitz continuity or convexity, both topics that we will cover later. As we discuss different algorithms, we will outline their assumptions, the motivation for their mechanism, and their advantages and disadvantages.

[19] The assumptions and results of the no free lunch theorems are provided by D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

1.4 Applications

The optimization algorithms discussed in this book have been applied to a wide variety of problems spanning many disciplines. This section discusses a few conceptual examples with real-world applications.

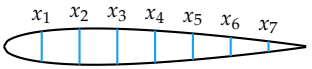


Figure 1.6. A symmetric airfoil parameterized by the design variables $x_1, \ldots, x_7$.

1.4.1 Aircraft Design

Aircraft design is a complex optimization problem that involves many disciplines, ranging from aerodynamics to structural mechanics. We can parameterize the design of the various components of an aircraft, such as the airfoil, with a set of design variables. For example, an airfoil can be parameterized by the thickness at various points along the chord length. Figure 1.6 shows a parameterization involving seven design variables. We want to search the space of design variables to find the values that minimize the drag of the airfoil while maintaining lift. In addition, we may have constraints such as the airfoil must have a certain minimum thickness so that it is structurally sound. To solve this optimization problem, we can use an optimization algorithm that searches the seven-dimensional design space to find an optimal airfoil shape that adheres to our constraints.

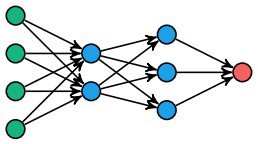


Figure 1.7. A deep neural network with two hidden layers. The input layer is shown in green, the hidden layers in blue, and the output layer in red. The parameters of the network are the weights on the connections between neurons and the associated biases added before the nonlinear activation function. In a home price estimation task, the inputs might correspond to the number of bedrooms, square footage, and year it was built, and the output would be the estimated price. The objective function would measure how well the network predicts the price of homes in the dataset. A review of deep learning is provided by I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

1.4.2 Deep Learning

Deep learning has been used to solve a wide variety of problems, such as image recognition, natural language processing, and game playing. In these applications, deep learning involves training a deep neural network to optimize an objective

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

function that measures how well it does on a task, such as predicting the sale price of a home given a number of attributes of that home. A neural network is simply a collection of neurons that are connected to each other. Each neuron combines a set of inputs by employing a linear function using a set of weights and a bias and then applies a nonlinear activation function to the result. These neurons are typically organized in layers, with each layer of neurons producing outputs that are fed to the next layer of neurons. When there are many layers, the network is called deep. The weights and biases of the network are the design variables that are optimized to perform well on the training set. Most neural network representations used in practice lend themselves well to computing the gradient of the objective function with respect to the weights and biases. Gradient information is often used by neural network training algorithms to help guide the optimization process.

1.4.3 *Statistical Modeling*

An important area of statistics involves inference of probabilistic models from observed data. One common approach is to define a parameterized probability distribution and then optimize the parameters of the distribution to best fit the data. The Gaussian mixture model is a type of parametric model defined by a weighted sum of Gaussian distributions, where each component has its own mean and standard deviation. Figure 1.8 shows a Gaussian mixture model with two components. The parameters of the model are optimized to maximize the likelihood of the observed data subject to the constraints that the weights are nonnegative and sum to one and that the standard deviations are positive. Although the optimal parameters of a single Gaussian distribution have a closed-form solution, finding the optimal parameters of a Gaussian mixture model generally requires an iterative optimization algorithm.

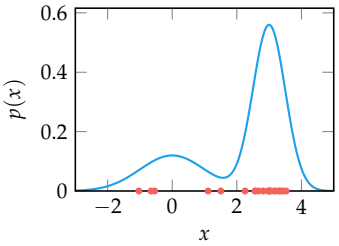


Figure 1.8. A Gaussian mixture model with two components. The red points correspond to observed data. The blue curve represents the probability density function $p(x)$ of the mixture model. Optimizing the parameters of this density function is discussed by K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

1.4.4 *Financial Portfolio Construction*

A major area of finance is portfolio construction. We have a budget and need to decide how to allocate our funds, such as to stocks, bonds, and other assets. The design variables correspond to what fraction of our budget we allocate to each asset, and they are constrained to be nonnegative and sum to one. The objective function might be to maximize the expected return of the portfolio while

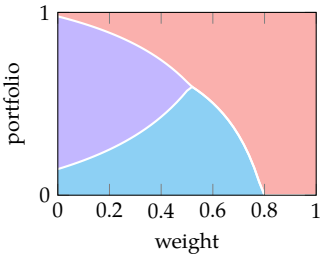


Figure 1.9. A portfolio optimization problem with three stocks (blue, purple, and red). The horizontal axis corresponds to how much we weight expected return versus risk, or the variance of the portfolio return. The vertical axis indicates what fraction of the budget is allocated to each stock in an optimal portfolio for a given risk-return weight. This kind of portfolio optimization problem is discussed in more detail in example 20.6.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

minimizing the risk. The risk of a portfolio is often measured by the variance of the return. We are uncertain about the return of each asset, but we can estimate the mean and covariance of the returns from past data. We can compute a scalar objective function that weights together the expected return and the risk of the portfolio, and then use an algorithm to find the optimal allocation of funds.

1.4.5 *Medical Radiotherapy*

Medical radiotherapy involves treating cancer patients with radiation to kill cancer cells while minimizing damage to healthy tissue. The radiation from a particle accelerator is often delivered in the form of an external ionizing radiation beam, which passes through both the tumor and the surrounding healthy tissue as shown in figure 1.10. These beams can be positioned and shaped, and their intensity can be modulated. There are many ways to formulate radiotherapy planning as an optimization problem. One way is to have the design variables correspond to the positions, shapes, and intensities of the beams. The objective function to be minimized is the amount of radiation that is delivered to the healthy tissue through the treatment plan. The dosage of radiation to the tumor may be constrained to be above some threshold. Some formulations of this problem are convex, which means that the optimization problem can be solved exactly and efficiently.

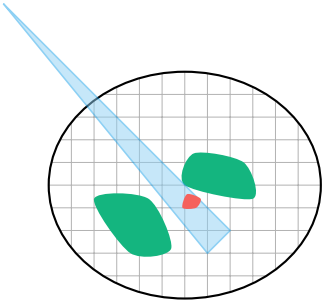


Figure 1.10. A medical radiotherapy problem. Shown is a cross section of the patient. The green blobs represent healthy organs, and the red blob represents a tumor. The blue wedge is a radiation beam. An optimization approach to treatment planning is presented by A. Fu, B. Ungun, L. Xing, and S. Boyd, "A Convex Optimization Approach to Radiation Treatment Planning with Dose Constraints," *Optimization and Engineering*, vol. 20, pp. 277–300, 2019.

1.4.6 *Robotic Planning*

Robotic planning involves finding a sequence of actions that a robot must take to achieve a goal. Figure 1.11 shows a simple example of a robot planning problem where the robot must navigate from a start position to a goal position while avoiding obstacles in a grid. The robot may go up, down, left, or right. The objective is to minimize the number of moves the robot makes to reach the goal. In contrast with the previous examples in this section, the design space is discrete. The robot can only move in one of four directions, and the design variables are the sequence of moves that the robot makes. Dynamic programming is a common optimization technique used to solve this type of problem efficiently.

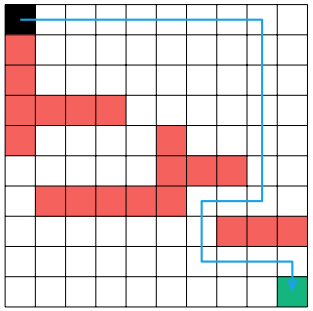


Figure 1.11. A robot planning problem. The robot starts at the black square and must reach the green square with the minimum number of steps. The robot can move up, down, left, or right, but it cannot move diagonally. The design variables are the sequence of moves that the robot makes. The red squares are obstacles that the robot must avoid. Section 22.5 discusses how dynamic programming can be used to solve this type of problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

1.4.7 Industrial Job Shop Scheduling

A classic problem in industrial operations research is job shop scheduling, where we need to allocate resources to jobs to minimize the total time it takes to process all jobs. We have a set of jobs and a set of machines, and each job must be processed on different machines in a specific order. The time required for each job on each machine is known ahead of time, and we want to find the order in which to process the jobs on the machines to minimize the total time to process all jobs. Figure 1.12 shows an example of a job shop schedule with four machines and three jobs. There is no known polynomial-time algorithm to solve this problem exactly, but there are many heuristics and optimization algorithms that can be used to find good solutions.

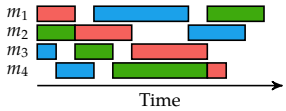


Figure 1.12. An industrial job shop scheduling problem. There are four machines, $m_1, \ldots, m_4$, and there are three jobs indicated by red, blue, and green. The objective is to minimize the total time it takes to process all jobs. There are many variations of this problem as reviewed by H. Xiong, S. Shi, D. Ren, and J. Hu, "A Survey of Job Shop Scheduling Problem: The Types and Models," *Computers and Operations Research*, vol. 142, p. 105731, 2022.

1.5 Minima

When minimizing f, we wish to find a *global minimizer*, a value of x for which $f(x)$ is minimized. A function may have at most one global minimum, but it may have multiple global minimizers. Unfortunately, it is generally difficult to prove that a given candidate point is at a global minimum. Often, the best we can do is check whether it is at a *local minimum*. When f is a *univariate function*, a point x^* is at a local minimum (or is a local minimizer) if there exists a $\delta > 0$ such that $f(x^*) \leq f(x)$ for all x with $|x - x^*| < \delta$. When f is a *multivariate function*, this definition generalizes to there being a $\delta > 0$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ whenever $\|\mathbf{x} - \mathbf{x}^*\| < \delta$.

Figure 1.13 shows two types of local minima: *strong local minima* and *weak local minima*. A *strong local minimizer*, also known as a *strict local minimizer*, is a point that uniquely minimizes f within a neighborhood. In other words, x^* is a strict local minimizer if there exists a $\delta > 0$ such that $f(x^*) < f(x)$ whenever $x^* \neq x$ and $|x - x^*| < \delta$. In the multivariate context, this generalizes to there being a $\delta > 0$ such that $f(\mathbf{x}^*) < f(\mathbf{x})$ whenever $\mathbf{x}^* \neq \mathbf{x}$ and $\|\mathbf{x} - \mathbf{x}^*\| < \delta$. A weak local minimizer is a local minimizer that is not a strong local minimizer.

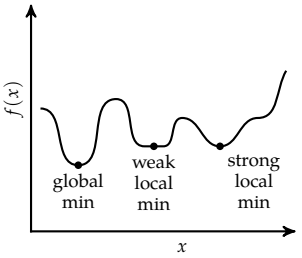


Figure 1.13. Examples of different types of minima.

1.6 Optimality Conditions

In cases where the objective function is twice differentiable (at least at the minimizer),[20] we can establish conditions for optimality. Our discussion in this section

[20] Differentiation is reviewed in the next chapter.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

assumes that the problem is unconstrained. Conditions for optimality in constrained problems are introduced in chapter 10. We will first discuss the conditions for local minima in univariate functions, and then extend these conditions to multivariate functions, where there are multiple design variables.

1.6.1 Univariate Conditions

A design point is guaranteed to be at a strong local minimum if the local derivative is zero and the second derivative is positive:

1. $f'(x^*) = 0$
2. $f''(x^*) > 0$

A zero derivative ensures that shifting the point by small values does not significantly affect the function value. A positive second derivative ensures that the zero first derivative occurs at the bottom of a *bowl*.[21]

[21] If $f'(x) = 0$ and $f''(x) < 0$, then x is a local maximum.

A point can also be at a local minimum if it has a zero derivative and the second derivative is merely nonnegative:

1. $f'(x^*) = 0$, the *first-order necessary condition*[22]
2. $f''(x^*) \geq 0$, the *second-order necessary condition*

[22] A point that satisfies the first-order necessary condition is sometimes called a *stationary point*.

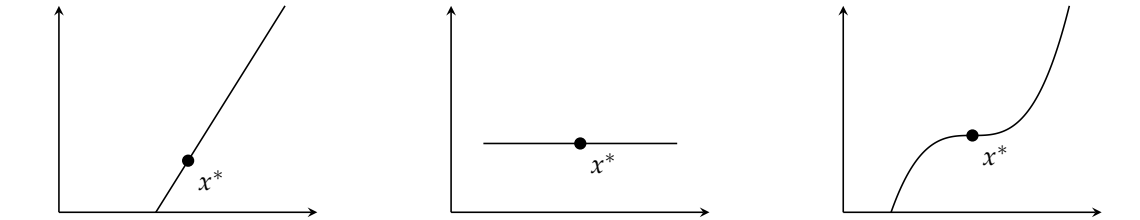
These conditions are referred to as *necessary* because all local minima obey these two rules. Unfortunately, not all points with a zero derivative and a zero second derivative are local minima, as demonstrated in figure 1.14.

The first-order necessary condition can be derived using the Taylor expansion[23] about our candidate point x^*:

[23] The Taylor expansion is derived in appendix C.

$$f(x^* + h) = f(x^*) + hf'(x^*) + O(h^2) \tag{1.4}$$
$$f(x^* - h) = f(x^*) - hf'(x^*) + O(h^2) \tag{1.5}$$
$$f(x^* + h) \geq f(x^*) \implies hf'(x^*) \geq 0 \tag{1.6}$$
$$f(x^* - h) \geq f(x^*) \implies hf'(x^*) \leq 0 \tag{1.7}$$
$$\implies f'(x^*) = 0 \tag{1.8}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



second-order but not first-order | first-order and second-order | first-order and second-order

Figure 1.14. Examples of the necessary but insufficient conditions for strong local minima.

where the asymptotic notation $O(h^2)$ is reviewed in appendix C.

The second-order necessary condition can also be obtained from the Taylor expansion:

$$f(x^* + h) = f(x^*) + \underbrace{hf'(x^*)}_{=0} + \frac{h^2}{2}f''(x^*) + O(h^3) \tag{1.9}$$

We know that the first-order necessary condition must apply:

$$f(x^* + h) \geq f(x^*) \implies \frac{h^2}{2}f''(x^*) \geq 0 \tag{1.10}$$

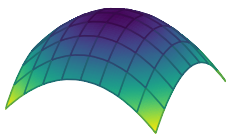
since $h > 0$. It follows that $f''(x^*) \geq 0$ must hold for x^* to be at a local minimum.

1.6.2 *Multivariate Conditions*

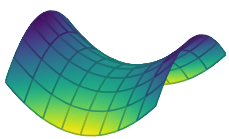
We can generalize the necessary conditions for local minima to multivariate functions as follows:

1. $\nabla f(\mathbf{x}) = \mathbf{0}$, the first-order necessary condition.
2. $\nabla^2 f(\mathbf{x})$ is positive semidefinite (for a review of this definition, see appendix C.7), the second-order necessary condition.

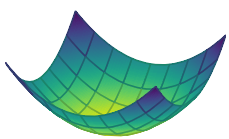
The first-order necessary condition tells us that the function is not changing at $\mathbf{x}$. Figure 1.15 shows examples of multivariate functions where the first-order necessary condition is satisfied. The second-order necessary condition tells us that $\mathbf{x}$ is in a bowl. Example 1.1 provides an example of a way to visualize such a function.



A *hill*. The gradient at the center is zero, but the Hessian is negative definite.



A *saddle*. The gradient at the center is zero, but it is not a local minimum.



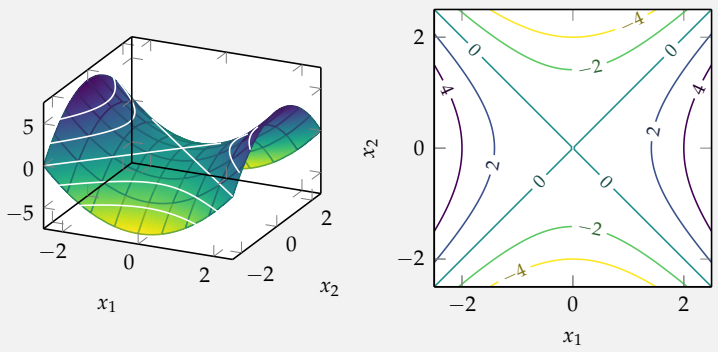
A *bowl*. The gradient at the center is zero and the Hessian is positive definite. It is a local minimum.

Figure 1.15. The three local regions where the gradient is zero.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 1.1. An example of three-dimensional visualization and the associated contour plot.

Most of the examples in this book include one or two design variables to facilitate plotting, but many real-world problems have more. For two-dimensional problems, a function of the form $f(x_1, x_2) = y$ can be rendered in three-dimensional space. A *contour plot* is a visual representation of a three-dimensional surface obtained by plotting regions with constant $y = f(\mathbf{x})$ values, known as *contours*, on a two-dimensional plot with axes indexed by x_1 and x_2. The plots below show $f(x_1, x_2) = x_1^2 - x_2^2$ as both a three-dimensional surface and a contour plot.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

These conditions can be obtained from a simple analysis. In order for $\mathbf{x}^*$ to be at a local minimum, it must be smaller than those values around it:

$$f(\mathbf{x}^*) \leq f(\mathbf{x}^* + h\mathbf{y}) \quad \Leftrightarrow \quad f(\mathbf{x}^* + h\mathbf{y}) - f(\mathbf{x}^*) \geq 0 \tag{1.11}$$

If we write the second-order approximation for $f(\mathbf{x}^*)$, we get:

$$f(\mathbf{x}^* + h\mathbf{y}) = f(\mathbf{x}^*) + h\nabla f(\mathbf{x}^*)^\top \mathbf{y} + \frac{1}{2}h^2 \mathbf{y}^\top \nabla^2 f(\mathbf{x}^*)\mathbf{y} + O(h^3) \tag{1.12}$$

We know that at a minimum, the first derivative must be zero, and we neglect the higher-order terms. Rearranging, we get:

$$\frac{1}{2}h^2 \mathbf{y}^\top \nabla^2 f(\mathbf{x}^*)\mathbf{y} = f(\mathbf{x}^* + h\mathbf{y}) - f(\mathbf{x}^*) \geq 0 \tag{1.13}$$

This is the definition of a positive semidefinite matrix, and we recover the second-order necessary condition. Example 1.2 illustrates how these conditions can be applied to the Rosenbrock banana function.

While necessary for optimality, these conditions are not sufficient for optimality. For unconstrained optimization of a twice-differentiable function, a point is guaranteed to be at a strong local minimum if the first-order necessary condition is satisfied and $\nabla^2 f(\mathbf{x})$ is positive definite. These conditions are collectively known as the *second-order sufficient condition*.

1.7 Overview

This section provides a brief overview of the chapters of this book. The conceptual dependencies between the chapters are outlined in figure 1.16.

Chapter 2 begins by discussing *derivatives* and their generalization to multiple dimensions. Derivatives are used in many algorithms to inform the choice of direction of the search for an optimum. Often derivatives are not known analytically, and so we discuss how to estimate them numerically and using automatic differentiation techniques.

Chapter 3 discusses *bracketing*, which involves identifying an interval in which a local minimum lies for a univariate function. Different bracketing algorithms use different schemes for successively shrinking the interval based on function evaluations. We can use knowledge about the function, such as its Lipschitz constant, to guide the bracketing process. These bracketing algorithms are often used as subroutines within the optimization algorithms discussed later in the text.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 1.2. Checking the first- and second-order necessary conditions of a point on the Rosenbrock function. The minimizer is indicated by the dot in the figure below.

Consider the Rosenbrock banana function,

$$f(\mathbf{x}) = (1 - x_1)^2 + 5(x_2 - x_1^2)^2$$

Does the point $[1, 1]$ satisfy the first-order and second-order necessary conditions?

The gradient is:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2(10x_1^3 - 10x_1x_2 + x_1 - 1) \\ 10(x_2 - x_1^2) \end{bmatrix}$$

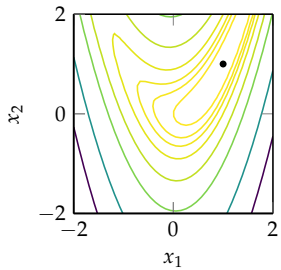
and the Hessian is:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} \end{bmatrix} = \begin{bmatrix} -20(x_2 - x_1^2) + 40x_1^2 + 2 & -20x_1 \\ -20x_1 & 10 \end{bmatrix}$$

We compute $\nabla f(1, 1) = 0$, so the first-order necessary condition is satisfied. The Hessian at $[1, 1]$ is:

$$\begin{bmatrix} 42 & -20 \\ -20 & 10 \end{bmatrix}$$

which is positive definite, so the second-order necessary condition is also satisfied.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

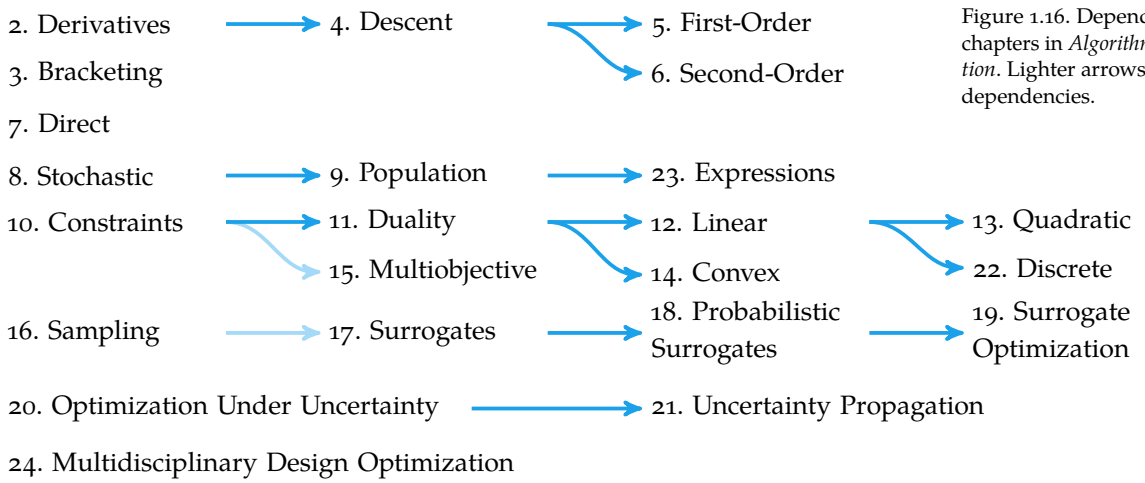


Figure 1.16. Dependencies for the chapters in *Algorithms for Optimization*. Lighter arrows indicate weak dependencies.

Chapter 4 introduces *local descent* as a general approach to optimizing multivariate functions. Local descent involves iteratively choosing a descent direction and then taking a step in that direction and repeating that process until convergence or some termination condition is met. There are different schemes for choosing the step length. We will also discuss methods that adaptively restrict the step size to a region where there is confidence in the local model.

Chapter 5 builds on the previous chapter, explaining how to use *first-order* information obtained through the gradient estimate as a local model to inform the descent direction. Simply stepping in the direction of steepest descent is often not the best strategy for finding a minimum. This chapter discusses a wide variety of different methods for using the past sequence of gradient estimates to better inform the search.

Chapter 6 shows how to use local models based on *second-order* approximations to inform local descent. These models are based on estimates of the Hessian of the objective function. The advantage of second-order approximations is that they can inform both the direction and step size.

Chapter 7 presents a collection of *direct methods* for finding optima that avoid using gradient information for informing the direction of search. We begin by discussing methods that iteratively perform line search along a set of directions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We then discuss pattern search methods that do not perform line search but rather perform evaluations some step size away from the current point along a set of directions. The step size is incrementally adapted as the search proceeds. Another method uses a simplex that adaptively expands and contracts as it traverses the design space in the apparent direction of improvement. Finally, we discuss a method motivated by Lipschitz continuity to increase resolution in areas deemed likely to contain the global minimum.

Chapter 8 introduces *stochastic* methods, where randomness is incorporated into the optimization process. We show how stochasticity can improve some of the algorithms discussed in earlier chapters, such as steepest descent and pattern search. Some of the methods involve incrementally traversing the search space, but others involve learning a probability distribution over the design space, assigning greater weight to regions that are more likely to contain an optimum.

Chapter 9 discusses *population* methods, where a collection of points is used to explore the design space. Having a large number of points distributed through the space can help reduce the risk of becoming stuck in a local minimum. Population methods generally rely on stochasticity to encourage diversity in the population, and they can be combined with local descent methods.

Chapter 10 introduces the notion of *constraints* in optimization problems. In some cases, we can transform a problem with constraints into an unconstrained optimization problem. In other cases, we might want to use the concept of *Lagrange multipliers* to define mathematical conditions for optimality. For general optimization problems, we can use *penalty methods*, which incorporate constraints into the optimization objective to penalize infeasibility. By moving the constraints into the objective, we can use the various unconstrained optimization methods discussed earlier. We also discuss methods for ensuring that, if we start with a feasible point, the search will remain feasible.

Chapter 11 discusses the concept of *duality* in optimization problems. Duality provides a way to transform a constrained optimization problem into another optimization problem called the *dual problem* that can be easier to solve than the original or *primal problem* and provides a lower bound on the primal solution. For some classes of problems, the dual solution is the same as the primal solution. We discuss optimization algorithms that alternate between the primal and dual problems to find the optimal solution.

Chapter 12 makes the assumption that both the objective function and constraints are *linear*. Although linearity may appear to be a strong assumption, many

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

engineering problems can be framed as linear-constrained optimization problems. Several methods have been developed for exploiting this linear structure. This chapter focuses on the simplex algorithm, which is guaranteed to result in a global minimum.

Chapter 13 discusses the special case where the objective function is *quadratic* and the constraints are linear. We show how convex quadratic optimization problems can be transformed into what is known as a *least-squares problem*. Unconstrained least-squares problems can be solved exactly using the pseudoinverse. General least-squares problems can be transformed into least-squares problems with linear inequalities, which can then be solved by transforming them into nonnegative least-squares problems.

Chapter 14 discusses the concept of *disciplined convex programming*. Convex problems are those with a convex objective function and constraints that define a convex feasible set. Convex optimization problems have a single global minimum, and many methods have been developed to efficiently compute a solution. We introduce the concept of a disciplined convex program, which is a convex optimization problem that can be written in such a way that automated methods can both verify that the problem is convex and automatically transcribe the problem into a canonical form.

Chapter 15 shows how to address the problem of *multiobjective* optimization, where we have multiple objectives that we are trying to optimize simultaneously. Engineering often involves a trade-off between multiple objectives, and it is often unclear how to prioritize different objectives. We discuss how to transform multiobjective problems into scalar-valued objective functions so that we can use the algorithms discussed in earlier chapters. We also discuss algorithms for finding the set of design points that represent the best trade-off between objectives.

Chapter 16 discusses how to create *sampling plans* consisting of points that cover the design space. Random sampling of the design space often does not provide adequate coverage. We discuss methods for ensuring uniform coverage along each design dimension and methods for measuring and optimizing the coverage of the space. In addition, we discuss quasi-random sequences that can also be used to generate sampling plans.

Chapter 17 explains how to build *surrogate models* of the objective function. Surrogate models are often used for problems where evaluating the objective function is very expensive. An optimization algorithm can then use evaluations of the surrogate model instead of the actual objective function to improve the design.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The evaluations can come from historical data, perhaps obtained through the use of a sampling plan introduced in the previous chapter. We discuss different types of surrogate models, how to fit them to data, and how to identify a suitable surrogate model.

Chapter 18 introduces *probabilistic surrogate models* that allow us to quantify our confidence in the predictions of the models. This chapter focuses on a particular type of surrogate model called a Gaussian process. We show how to use Gaussian processes for prediction, how to incorporate gradient measurements and noise, and how to estimate some of the parameters governing the Gaussian process from data.

Chapter 19 shows how to use the probabilistic models from the previous chapter to guide *surrogate optimization*. The chapter outlines several techniques for choosing which design point to evaluate next. We also discuss how surrogate models can be used to optimize an objective measure in a safe manner.

Chapter 20 explains how to perform *optimization under uncertainty*, relaxing the assumption made in previous chapters that the objective function is a deterministic function of the design variables. We discuss different approaches for representing uncertainty, including set-based and probabilistic approaches, and explain how to transform the problem to provide robustness to uncertainty.

Chapter 21 outlines approaches to *uncertainty propagation*, where known input distributions are used to estimate statistical quantities associated with the output distribution. Understanding the output distribution of an objective function is important to optimization under uncertainty. We discuss a variety of approaches, some based on mathematical concepts such as Monte Carlo, the Taylor series approximation, orthogonal polynomials, and Gaussian processes. They differ in the assumptions they make and the quality of their estimates.

Chapter 22 shows how to approach problems where the design variables are constrained to be *discrete*. A common approach is to relax the assumption that the variables are discrete, but this can result in infeasible designs. Another approach involves incrementally adding linear constraints until the optimal point is discrete. We also discuss branch and bound along with dynamic programming approaches, both of which guarantee optimality. The chapter also mentions a population-based method that often scales to large design spaces but does not provide guarantees.

Chapter 23 discusses how to search design spaces consisting of *expressions* defined by a grammar. For many problems, the number of variables is unknown, such as in the optimization of graphical structures or computer programs. We

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

outline several algorithms that account for the grammatical structure of the design space to make the search more efficient.

Chapter 24 explains how to approach *multidisciplinary design optimization*. Many engineering problems involve complicated interactions between several disciplines, and optimizing disciplines individually may not lead to an optimal solution. This chapter discusses a variety of techniques for taking advantage of the structure of multidisciplinary problems to reduce the effort required for finding good designs.

The appendices contain supplementary material. Appendix A begins with a short introduction to the Julia programming language, focusing on the concepts used to specify the algorithms listed in this book. Appendix B specifies a variety of test functions used for evaluating the performance of different algorithms. Appendix C covers mathematical concepts used in the derivation and analysis of the optimization methods discussed in this text.

1.8 Summary

- Optimization in engineering is the process of finding the best system design subject to a set of constraints.
- Algorithms for optimization can be applied to a wide variety of fields, ranging from aircraft design to deep learning.
- Minima occur where the gradient is zero, but zero-gradient does not imply optimality.

1.9 Exercises

Exercise 1.1. Give an example of a function with a local minimum that is not a global minimum.

Solution: $f(x) = x^3/3 - x$ at $x = 1$.

Exercise 1.2. What is the minimum and minimizer of the function $f(x) = x^3 - x$?

Solution: It does not have a minimum. The function is said to be unbounded below.

Exercise 1.3. Give an example of a function that is bounded below but has no global optima when the feasible set is the interval $[0, 1]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: Here is one example:

$$f(x) = \begin{cases} |x| & \text{if } x \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

Exercise 1.4. What are the minima and minimizers of the function $f(x) = \sin(x)$?

Solution: The minimum value is -1, which is obtained by the minimizers $x = \frac{3}{2}\pi + 2\pi i$ for all integers i.

Exercise 1.5. Does the first-order condition $f'(x) = 0$ hold when x is the optimal solution of a constrained problem?

Solution: No. Consider minimizing $f(x) = x$, subject to $x \geq 1$.

Exercise 1.6. How many minima does $f(x, y) = x^2 + y$, subject to $x > y \geq 1$, have?

Solution: The function f can be broken into two separate functions that depend only on their specific coordinate:

$$f(x, y) = g(x) + h(y)$$

where $g(x) = x^2$ and $h(y) = y$. Both g and h strictly increase for $x, y \geq 1$. While h is minimized for $y = 1$, we can merely approach $x \to 1$ due to the strict inequality $x > y$. Thus, f has no minima.

Exercise 1.7. How many inflection points does $f(x) = x^3 - 10$ have?

Solution: An inflection point is a point on a curve where the sign of the curvature changes. When f is continuously twice-differentiable, a necessary condition for x to be an inflection point on f is that the second derivative is zero. In this case, f is continuously twice-differentiable. The second derivative is $f''(x) = 6x$, which is only zero at $x = 0$. A sufficient condition for x to be an inflection point is that the second derivative changes sign around x. That is, $f''(x + \epsilon)$ and $f''(x - \epsilon)$ for $\epsilon \ll 1$ have opposite signs. This holds for $x = 0$, so it is an inflection point. There is thus only one inflection point on $x^3 - 10$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

2 Derivatives and Gradients

Optimization is concerned with finding the design point that minimizes (or maximizes) an objective function. Knowing how the value of a function changes as its input is varied is useful because it tells us in which direction we can move to improve on previous points. The change in the value of the function is measured by the derivative in one dimension and the gradient in multiple dimensions. This chapter briefly reviews some essential elements from calculus.[1]

[1] For a more comprehensive review, see S. J. Colley, *Vector Calculus*, 4th ed. Pearson, 2011.

2.1 Derivatives

The *derivative* $f'(x)$ of a function f of a single variable x is the rate at which the value of f changes at x. It is often visualized, as shown in figure 2.1, using the tangent line to the graph of the function at x. The value of the derivative equals the slope of the tangent line.

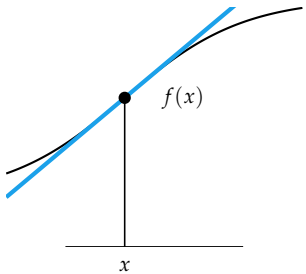


Figure 2.1. The function f is drawn in black and the tangent line to $f(x)$ is drawn in blue. The derivative of f at x is the slope of the tangent line.

We can use the derivative to provide a linear approximation of the function near x:

$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x \tag{2.1}$$

The derivative is the ratio between the change in f and the change in x at the point x:

$$f'(x) = \frac{\Delta f(x)}{\Delta x} \tag{2.2}$$

which is the change in $f(x)$ divided by the change in x as the step becomes infinitesimally small as illustrated by figure 2.2.

The notation $f'(x)$ can be attributed to Lagrange. We also use the notation created by Leibniz,

$$f'(x) \equiv \frac{df(x)}{dx} \tag{2.3}$$

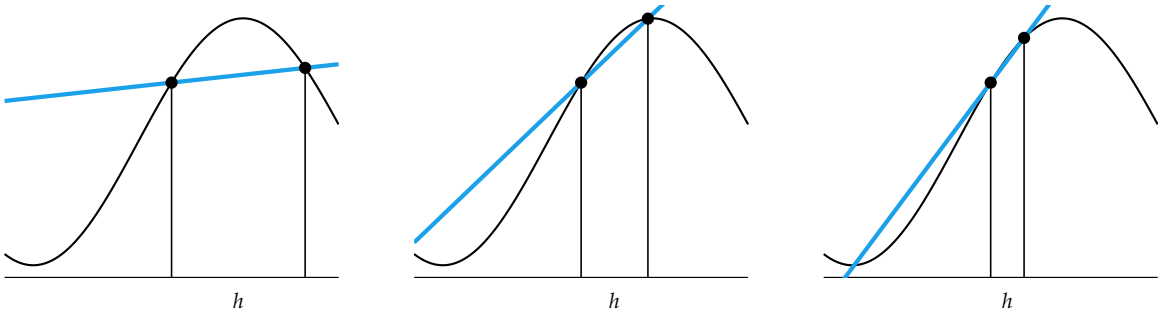


Figure 2.2. The tangent line is obtained by joining points with sufficiently small step differences.

which emphasizes the fact that the derivative is the ratio of the change in f to the change in x at the point x.

The limit equation defining the derivative can be presented in three different ways: the *forward difference*, the *central difference*, and the *backward difference*. Each method uses an infinitely small step size h:

$$f'(x) \equiv \underbrace{\lim_{h\to 0} \frac{f(x+h) - f(x)}{h}}_{\text{forward difference}} = \underbrace{\lim_{h\to 0} \frac{f(x+h/2) - f(x-h/2)}{h}}_{\text{central difference}} = \underbrace{\lim_{h\to 0} \frac{f(x) - f(x-h)}{h}}_{\text{backward difference}} \tag{2.4}$$

If f can be represented symbolically, *symbolic differentiation* can often provide an exact analytic expression for f' by applying derivative rules from calculus. The analytic expression can then be evaluated at any point x. The process is illustrated in example 2.1.

Example 2.1. Symbolic differentiation provides analytical derivatives.

The implementation details of symbolic differentiation are outside the scope of this text. Various software packages, such as `SymEngine.jl` in Julia and `SymPy` in Python, provide implementations. Here we use `SymEngine.jl` to compute the derivative of $x^2 + x/2 - \sin(x)/x$.

```
julia> using SymEngine
julia> @vars x; # define x as a symbolic variable
julia> f = x^2 + x/2 - sin(x)/x;
julia> diff(f, x)
1/2 + 2*x + sin(x)/x^2 - cos(x)/x
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

2.2 *Derivatives in Multiple Dimensions*

The *gradient* is the generalization of the derivative to multivariate functions. It captures the local slope of the function, allowing us to predict the effect of taking a small step from a point in any direction. Recall that the derivative is the slope of the tangent line. The gradient points in the direction of steepest ascent of the tangent *hyperplane* as shown in figure 2.3. A hyperplane in an n-dimensional space is the set of points that satisfies

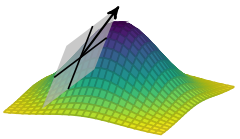


Figure 2.3. Each component of the gradient defines a local tangent line. These tangent lines define the local tangent hyperplane. The gradient vector points in the direction of greatest increase.

$$w_1 x_1 + \cdots + w_n x_n = b \tag{2.5}$$

for some vector $\mathbf{w}$ and scalar b. A hyperplane has $n - 1$ dimensions.

The gradient of f at $\mathbf{x}$ is written $\nabla f(\mathbf{x})$ and is a vector. Each component of that vector is the *partial derivative*[2] of f with respect to that component:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \quad \frac{\partial f(\mathbf{x})}{\partial x_2}, \quad \ldots, \quad \frac{\partial f(\mathbf{x})}{\partial x_n}\right] \tag{2.6}$$

[2] The partial derivative of a function with respect to a variable is the derivative assuming all other input variables are held constant. It is denoted $\partial f / \partial x$.

We use the convention that vectors written with commas are column vectors. For example, we have $[a, b, c] = [a \ b \ c]^\top$. Example 2.2 shows how to compute the gradient of a function at a particular point.

Example 2.2. Computing the gradient at a particular point.

Compute the gradient of $f(\mathbf{x}) = x_1 \sin(x_2) + 1$ at $\mathbf{c} = [2, 0]$.

$$\begin{aligned} f(\mathbf{x}) &= x_1 \sin(x_2) + 1 \\ \nabla f(\mathbf{x}) &= \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right] = [\sin(x_2), x_1 \cos(x_2)] \\ \nabla f(\mathbf{c}) &= [0, 2] \end{aligned}$$

The *Hessian* of a multivariate function is a matrix containing all of the second derivatives with respect to the input.[3] The second derivatives capture information about the local curvature of the function.

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ & \vdots & & \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_n} \end{bmatrix} \tag{2.7}$$

[3] The Hessian is symmetric if the second derivatives of f are all continuous in a neighborhood of the point at which it is being evaluated:

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = \frac{\partial^2 f}{\partial x_2 \partial x_1}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The *directional derivative* $\nabla_{\mathbf{s}} f(\mathbf{x})$ of a multivariate function f is the instantaneous rate of change of $f(\mathbf{x})$ as $\mathbf{x}$ is moved with velocity $\mathbf{s}$. The definition is closely related to the definition of a derivative of a univariate function:[4]

[4] Some texts require that $\mathbf{s}$ be a unit vector. See, for example, G. B. Thomas, *Calculus and Analytic Geometry*, 9th ed. Addison-Wesley, 1968.

$$\nabla_{\mathbf{s}} f(\mathbf{x}) \equiv \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}) - f(\mathbf{x})}{h}}_{\text{forward difference}} = \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{s}/2) - f(\mathbf{x} - h\mathbf{s}/2)}{h}}_{\text{central difference}} = \underbrace{\lim_{h \to 0} \frac{f(\mathbf{x}) - f(\mathbf{x} - h\mathbf{s})}{h}}_{\text{backward difference}} \tag{2.8}$$

The directional derivative can be computed using the gradient of the function:

$$\nabla_{\mathbf{s}} f(\mathbf{x}) = \nabla f(\mathbf{x})^\top \mathbf{s} \tag{2.9}$$

Another way to compute the directional derivative $\nabla_{\mathbf{s}} f(\mathbf{x})$ is to define $g(\alpha) \equiv f(\mathbf{x} + \alpha\mathbf{s})$ and then compute $g'(0)$, as illustrated in example 2.3.

The directional derivative is highest in the gradient direction, and it is lowest in the direction opposite the gradient. This directional dependence arises from the dot product in the directional derivative's definition and from the fact that the gradient is a local tangent hyperplane.

Example 2.3. Computing a directional derivative.

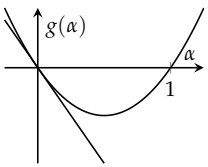
We wish to compute the directional derivative of $f(\mathbf{x}) = x_1 x_2$ at $\mathbf{x} = [1, 0]$ in the direction $\mathbf{s} = [-1, -1]$:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}, & \frac{\partial f}{\partial x_2} \end{bmatrix} = [x_2, x_1]$$

$$\nabla_{\mathbf{s}} f(\mathbf{x}) = \nabla f(\mathbf{x})^\top \mathbf{s} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix} = -1$$

We can also compute the directional derivative as follows:

$$g(\alpha) = f(\mathbf{x} + \alpha\mathbf{s}) = (1 - \alpha)(-\alpha) = \alpha^2 - \alpha$$
$$g'(\alpha) = 2\alpha - 1$$
$$g'(0) = -1$$



2.3 *Numerical Differentiation*

The process of estimating derivatives numerically is referred to as *numerical differentiation*. Estimates can be derived in different ways from function evaluations. This section discusses finite difference methods and the complex step method.[5]

[5] For a more comprehensive treatment of the topics discussed in the remainder of this chapter, see A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

2.3.1 *Finite Difference Methods*

As the name implies, *finite difference methods* compute the difference between two values that differ by a finite step size. They approximate the derivative definitions in equation (2.4) using small differences:

$$f'(x) \approx \underbrace{\frac{f(x+h) - f(x)}{h}}_{\text{forward difference}} \approx \underbrace{\frac{f(x+h/2) - f(x-h/2)}{h}}_{\text{central difference}} \approx \underbrace{\frac{f(x) - f(x-h)}{h}}_{\text{backward difference}} \tag{2.10}$$

Mathematically, the smaller the step size h, the better the derivative estimate. Practically, values for h that are too small can result in numerical cancellation errors. This effect is shown later in figure 2.4. Algorithm 2.1 provides implementations for these methods.

```
diff_forward(f, x; h=1e-9) = (f(x+h) - f(x))/h
diff_central(f, x; h=1e-9) = (f(x+h/2) - f(x-h/2))/h
diff_backward(f, x; h=1e-9) = (f(x) - f(x-h))/h
```

Algorithm 2.1. Finite difference methods for estimating the derivative of a function `f` at `x` with finite difference `h`. Good step sizes are on the order of the square root or cube root of the machine precision for floating point values. These step sizes balance machine round-off error with step size error.

The finite difference methods can be derived using the Taylor expansion. We will derive the forward difference derivative estimate, beginning with the Taylor expansion of f about x:

$$f(x+h) = f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \cdots \tag{2.11}$$

We can rearrange and solve for the first derivative:

$$f'(x)h = f(x+h) - f(x) - \frac{f''(x)}{2!}h^2 - \frac{f'''(x)}{3!}h^3 - \cdots \tag{2.12}$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(x)}{2!}h - \frac{f'''(x)}{3!}h^2 - \cdots \tag{2.13}$$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{2.14}$$

The forward difference approximates the true derivative for small h with error dependent on $\frac{f''(x)}{2!}h + \frac{f'''(x)}{3!}h^2 + \cdots$. This error term is $O(h)$, meaning the forward difference has linear error as h approaches zero.[6]

The central difference method has an error term of $O(h^2)$.[7] We can derive this error term using the Taylor expansion. The Taylor expansions about x for

[6] Asymptotic notation is covered in appendix C.

[7] J. H. Mathews and K. D. Fink, *Numerical Methods Using MATLAB*, 4th ed. Pearson, 2004.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$f(x + h/2)$ and $f(x - h/2)$ are:

$$f(x + h/2) = f(x) + f'(x)\frac{h}{2} + \frac{f''(x)}{2!}\left(\frac{h}{2}\right)^2 + \frac{f'''(x)}{3!}\left(\frac{h}{2}\right)^3 + \cdots \tag{2.15}$$

$$f(x - h/2) = f(x) - f'(x)\frac{h}{2} + \frac{f''(x)}{2!}\left(\frac{h}{2}\right)^2 - \frac{f'''(x)}{3!}\left(\frac{h}{2}\right)^3 + \cdots \tag{2.16}$$

Subtracting these expansions produces:

$$f(x + h/2) - f(x - h/2) \approx 2f'(x)\frac{h}{2} + \frac{2}{3!}f'''(x)\left(\frac{h}{2}\right)^3 \tag{2.17}$$

We rearrange to obtain:

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h} - \frac{f'''(x)h^2}{24} \tag{2.18}$$

which shows that the approximation has quadratic error.

2.3.2 *Complex Step Method*

We often run into the problem of needing to choose a step size h small enough to provide a good approximation but not too small so as to lead to numerical subtractive cancellation issues. The *complex step method* bypasses the effect of subtractive cancellation by using a single function evaluation. We evaluate the function once after taking a step in the imaginary direction.[8]

The Taylor expansion for an imaginary step is:

$$f(x + ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \cdots \tag{2.19}$$

Taking only the imaginary component of each side produces a derivative approximation:

$$\operatorname{Im}(f(x + ih)) = hf'(x) - h^3\frac{f'''(x)}{3!} + \cdots \tag{2.20}$$

$$\Rightarrow f'(x) = \frac{\operatorname{Im}(f(x + ih))}{h} + h^2\frac{f'''(x)}{3!} - \cdots \tag{2.21}$$

$$= \frac{\operatorname{Im}(f(x + ih))}{h} + O(h^2) \text{ as } h \to 0 \tag{2.22}$$

[8] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso, "The Complex-Step Derivative Approximation," *ACM Transactions on Mathematical Software*, vol. 29, no. 3, pp. 245–262, 2003. Special care must be taken to ensure that the implementation of f properly supports complex numbers as input. In particular, f must be analytic. Nonsmooth functions like `abs` and `norm` are not analytic.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

An implementation is provided by algorithm 2.2. The real part approximates $f(x)$ to within $O(h^2)$ as $h \to 0$:

$$\operatorname{Re}(f(x + ih)) = f(x) - h^2 \frac{f''(x)}{2!} + \cdots \tag{2.23}$$

$$\Rightarrow f(x) = \operatorname{Re}(f(x + ih)) + h^2 \frac{f''(x)}{2!} - \cdots \tag{2.24}$$

Thus, we can evaluate both $f(x)$ and $f'(x)$ using a single evaluation of f with complex arguments. Example 2.4 shows the calculations involved for estimating the derivative of a function at a particular point. Algorithm 2.2 implements the complex step method. Figure 2.4 compares the numerical error of the complex step method to the forward and central difference methods as the step size is varied.

```
diff_complex(f, x; h=1e-9) = imag(f(x + h*im)) / h
```

Algorithm 2.2. The complex step method for estimating the derivative of a function `f` at `x` with finite difference `h`.

Consider $f(x) = \sin(x^2)$. The function value at $x = \pi/2$ is approximately 0.624266 and the derivative is $\pi \cos(\pi^2/4) \approx -2.45425$. We can arrive at this using the complex step method:

```
julia> f = x -> sin(x^2);
julia> v = f(π/2 + 0.001im);
julia> real(v) # f(x)
0.6242698144866649
julia> imag(v)/0.001 # f'(x)
-2.4542516170381785
```

Example 2.4. The complex step method for estimating derivatives.

2.4 Automatic Differentiation

This section introduces algorithms for the numeric evaluation of derivatives of functions specified by a computer program. Key to these *automatic differentiation* techniques is the application of the chain rule:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx}(f \circ g)(x) = \frac{df}{dg}\frac{dg}{dx} \tag{2.25}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

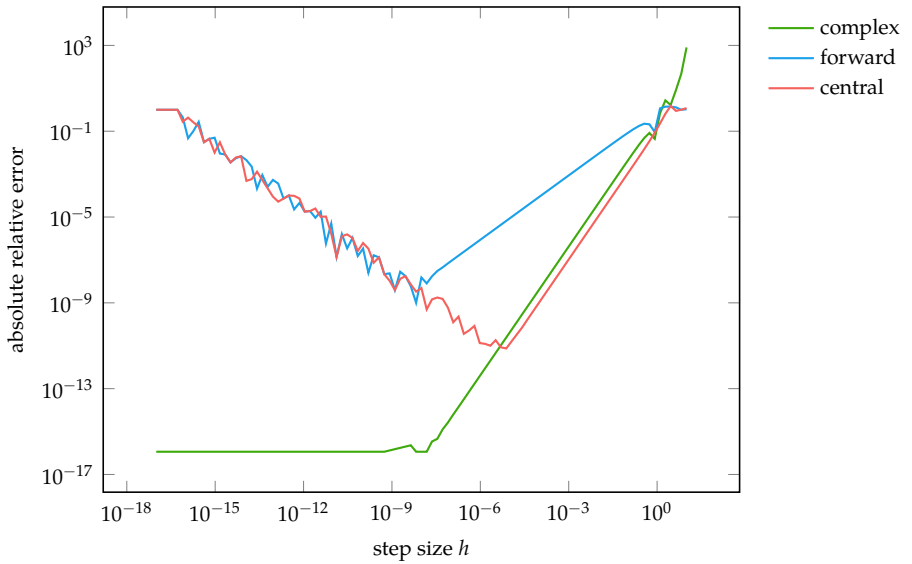


Figure 2.4. A comparison of the error in derivative estimate for the function $\sin(x^2)$ at $x = 1/2$ as the step size is varied. The linear error of the forward difference method and the quadratic error of the central difference and complex methods can be seen by the constant slopes on the right-hand side. The complex step method avoids the subtractive cancellation error that occurs when differencing two function evaluations that are close together.

A program is composed of elementary operations like addition, subtraction, multiplication, and division.

Consider the function $f(a, b) = \ln(ab + \max(a, 2))$. If we want to compute the partial derivative with respect to a at a point, we need to apply the chain rule several times:[9]

$$\frac{\partial f}{\partial a} = \frac{\partial}{\partial a} \ln(ab + \max(a, 2)) \tag{2.26}$$

$$= \frac{1}{ab + \max(a, 2)} \frac{\partial}{\partial a}(ab + \max(a, 2)) \tag{2.27}$$

$$= \frac{1}{ab + \max(a, 2)} \left[\frac{\partial (ab)}{\partial a} + \frac{\partial \max(a, 2)}{\partial a}\right] \tag{2.28}$$

$$= \frac{1}{ab + \max(a, 2)} \left[\left(b\frac{\partial a}{\partial a} + a\frac{\partial b}{\partial a}\right) + \left((2 > a)\frac{\partial 2}{\partial a} + (2 < a)\frac{\partial a}{\partial a}\right)\right] \tag{2.29}$$

$$= \frac{1}{ab + \max(a, 2)} [b + (2 < a)] \tag{2.30}$$

[9] We adopt the convention that Boolean expressions like $(2 < a)$ are 1 if true and 0 if false.

This process can be automated through the use of a *computational graph*. A computational graph represents a function where the nodes are operations and the edges are input-output relations. The leaf nodes of a computational graph

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

are input variables or constants, and terminal nodes are values output by the function. A computational graph is shown in figure 2.5.

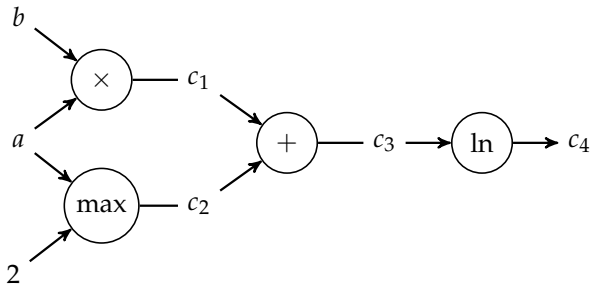


Figure 2.5. The computational graph for $\ln(ab + \max(a, 2))$.

There are two methods for automatically differentiating f using its computational graph. The *forward accumulation* method traverses the tree from inputs to outputs, whereas *reverse accumulation* requires a backward pass through the graph.

2.4.1 Forward Accumulation

Forward accumulation will automatically differentiate a function using a single forward pass through the function's computational graph. The method is equivalent to iteratively expanding the chain rule of the inner operation:

$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\frac{dc_3}{dx}\right) = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\left(\frac{dc_3}{dc_2}\frac{dc_2}{dx} + \frac{dc_3}{dc_1}\frac{dc_1}{dx}\right)\right) \tag{2.31}$$

To illustrate forward accumulation, we apply it to the example function $f(a, b) = \ln(ab + \max(a, 2))$ to calculate the partial derivative at $a = 3, b = 2$ with respect to a.

1. The procedure starts at the graph's source nodes consisting of the function inputs and any constant values. For each of these nodes, we note both the value and the partial derivative with respect to our target variable, as shown in figure 2.6.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

2. Next we proceed down the tree, one node at a time, choosing as our next node one whose inputs have already been computed. We can compute the value by passing through the previous nodes' values, and we can compute the local partial derivative with respect to a using both the previous nodes' values and their partial derivatives.

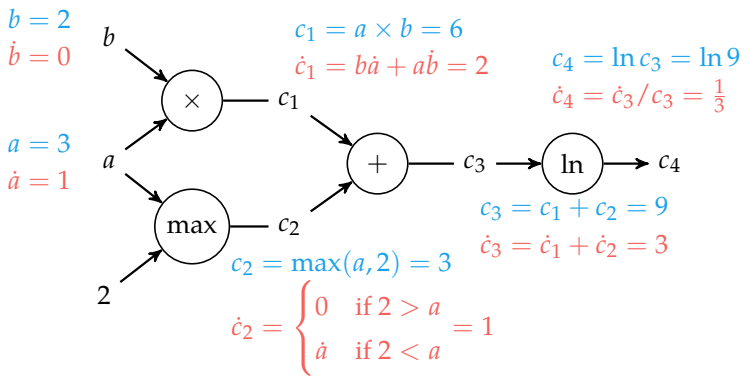


Figure 2.6. The computational graph for $\ln(ab + \max(a, 2))$ after forward accumulation is applied to calculate $\partial f / \partial a$ with $a = 3$ and $b = 2$. For compactness in this figure, we use *dot notation* or *Newton's notation* for derivatives. For example, if it is clear that we are taking the derivative with respect to a, we can write $\partial b / \partial a$ as $\dot{b}$. Because $\partial a / \partial a = 1$ and $\partial b / \partial a = 0$, we set $\dot{a} = 1$ and $\dot{b} = 0$ in this graph.

We end up with the correct result, $f(3, 2) = \ln 9$ and $\partial f / \partial a = 1/3$. This was done using one pass through the computational graph.

This process can be conveniently automated by a computer using a programming language which has overridden each operation to produce both the value and its derivative. Such pairs are called *dual numbers*.

Dual numbers can be expressed mathematically by including the abstract quantity ϵ, where ϵ^2 is defined to be 0. Like a complex number, a dual number is written $a + b\epsilon$, where a and b are both real values. In $a + b\epsilon$, a is the *real part* and $b\epsilon$ is the *dual part*. We have:

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon \tag{2.32}$$

$$(a + b\epsilon) \times (c + d\epsilon) = (ac) + (ad + bc)\epsilon \tag{2.33}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In fact, by passing a dual number into any smooth function f, we get the evaluation and its derivative. We can show this using the Taylor series:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k \tag{2.34}$$

$$f(a+b\epsilon) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(a+b\epsilon-a)^k \tag{2.35}$$

$$= \sum_{k=0}^{\infty} \frac{f^{(k)}(a)b^k\epsilon^k}{k!} \tag{2.36}$$

$$= f(a) + bf'(a)\epsilon + \epsilon^2 \sum_{k=2}^{\infty} \frac{f^{(k)}(a)b^k}{k!}\epsilon^{(k-2)} \tag{2.37}$$

$$= f(a) + bf'(a)\epsilon \tag{2.38}$$

Example 2.5 shows an implementation.

2.4.2 *Reverse Accumulation*

Forward accumulation requires n passes in order to compute an n-dimensional gradient. *Reverse accumulation*[10] requires only a single run in order to compute a complete gradient but requires two passes through the graph: a *forward pass* during which necessary intermediate values are computed and a *backward pass* that computes the gradient.[11] Reverse accumulation is often preferred over forward accumulation when gradients are needed, though care must be taken on memory-constrained systems when the computational graph is very large.[12]

Like forward accumulation, reverse accumulation will compute the partial derivative with respect to the chosen target variable but iteratively substitutes the outer function instead:

$$\frac{df}{dx} = \frac{df}{dc_1}\frac{dc_1}{dx} + \frac{df}{dc_2}\frac{dc_2}{dx} = \left(\left(\frac{df}{dc_4}\frac{dc_4}{dc_3}\right)\frac{dc_3}{dc_1}\right)\frac{dc_1}{dx} + \left(\left(\frac{df}{dc_4}\frac{dc_4}{dc_3}\right)\frac{dc_3}{dc_2}\right)\frac{dc_2}{dx} \tag{2.39}$$

This process is the reverse pass, the evaluation of which requires intermediate values that are obtained during a forward pass.

[10] S. Linnainmaa, "The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors," M.S. thesis, University of Helsinki, 1970.

[11] Evaluating the gradient of a function comprising a series of arithmetic operations and standard univariate functions like sin, log, and exp requires at most 5 times as many operations as merely evaluating the function. A. Griewank, "On Automatic Differentiation," *Mathematical Programming: Recent Developments and Applications*, pp. 83–108, 1989.

[12] Reverse accumulation is central to the backpropagation algorithm used to train neural networks. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533–536, 1986.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Dual numbers can be implemented by defining a struct `Dual` that contains two fields, the value `v` and the derivative `∂`.

```
struct Dual
    v # real part
    ∂ # dual part
end
```

We must then implement methods for each of the base operations required. These methods take in dual numbers and produce new dual numbers using that operation's chain rule logic.

```
Base.:+(a::Dual, b::Dual) = Dual(a.v + b.v, a.∂ + b.∂)
Base.:*(a::Dual, b::Dual) = Dual(a.v * b.v, a.v*b.∂ + b.v*a.∂)
Base.log(a::Dual) = Dual(log(a.v), a.∂/a.v)
Base.max(a::Dual, b::Dual) =
    Dual(max(a.v, b.v), a.v > b.v ? a.∂ : b.∂)
Base.max(a::Dual, b::Int) = Dual(max(a.v, b), a.v > b ? a.∂ : 0)
```

The `ForwardDiff.jl` package supports an extensive set of mathematical operations and additionally provides gradients and Hessians.

```
julia> using ForwardDiff
julia> a = ForwardDiff.Dual(3,1);
julia> b = ForwardDiff.Dual(2,0);
julia> log(a*b + max(a,2))
Dual{Nothing}(2.1972245773362196,0.3333333333333333)
```

Example 2.5. An implementation of dual numbers that allows for automatic forward accumulation. The package `DualNumbers.jl` provides comprehensive coverage of many additional base operations. While the gradient for max is not defined when the inputs are equal, this implementation returns the gradient of the second input instead of returning `NaN` or throwing an exception, an approach justified by the generalized gradient (appendix C.5).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Reverse accumulation can be implemented through *operation overloading*[13] in a similar manner to the way dual numbers are used to implement forward accumulation. Two functions must be implemented for each fundamental operation: a forward operation that overloads the operation to store local gradient information during the forward pass and a backward operation that uses the information to propagate the gradient backward. Packages like Tensorflow[14] or `Zygote.jl` can automatically construct the computational graph and the associated forward and backward pass operations. Example 2.6 shows how `Zygote.jl` can be used.

[13] Operation overloading refers to providing implementations for common operations such as $+$, $-$, or $=$ for custom variable types. Overloading is discussed in appendix A.2.5.

[14] Tensorflow is an open-source software library for numerical computation using data flow graphs and is often used for deep learning applications. It may be obtained from `tensorflow.org`.

Example 2.6. Automatic differentiation using the `Zygote.jl` package. We find that the gradient at $[3, 2]$ is $[1/3, 1/3]$.

The `Zygote.jl` package provides automatic differentiation in the form of reverse-accumulation. Here the `gradient` function is used to automatically generate the backward pass through the source code of `f` to obtain the gradient.

```
julia> import Zygote: gradient
julia> f(a, b) = log(a*b + max(a,2));
julia> gradient(f, 3.0, 2.0)
(0.3333333333333333, 0.3333333333333333)
```

2.5 Regression Gradient

Instead of estimating the gradient at $\mathbf{x}$ by using a finite difference method along each coordinate axis, we can use *linear regression*[15] to estimate the gradient from the results of random perturbations from $\mathbf{x}$.[16] More design perturbations will tend to produce better gradient estimates. This method is particularly useful when the objective function is noisy[17] because the regression helps smooth out the noise in the evaluations when producing a gradient estimate.

Given a dataset of m perturbations and their function evaluations

$$\left(\Delta\mathbf{x}^{(1)}, f(\mathbf{x} + \Delta\mathbf{x}^{(1)})\right), \left(\Delta\mathbf{x}^{(2)}, f(\mathbf{x} + \Delta\mathbf{x}^{(2)})\right), \ldots, \left(\Delta\mathbf{x}^{(m)}, f(\mathbf{x} + \Delta\mathbf{x}^{(m)})\right) \tag{2.40}$$

we seek the gradient $\mathbf{g}$ that is consistent with the first-order Taylor expansion:

$$\hat{f}(\mathbf{x} + \Delta\mathbf{x}) = f(\mathbf{x}) + \mathbf{g}^\top \Delta\mathbf{x} \tag{2.41}$$

[15] Linear regression is covered in section 17.2.

[16] This general approach is sometimes referred to as *simultaneous perturbation stochastic approximation* by J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley, 2003. The general connection to linear regression is provided by J. Peters and S. Schaal, "Reinforcement Learning of Motor Skills with Policy Gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008.

[17] Problems with noisy objective functions are covered in chapter 20.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We define the gradient with the closest match to be the one that minimizes the squared distance:

$$\underset{\mathbf{g}}{\text{minimize}} \sum_{i=1}^{m} \left(f(\mathbf{x} + \Delta\mathbf{x}^{(i)}) - \left(f(\mathbf{x}) + \mathbf{g}^\top \Delta\mathbf{x}^{(i)} \right) \right)^2 \tag{2.42}$$

We can solve this minimization exactly. To do so, we rewrite the optimization problem in matrix form, where we define

$$\Delta\mathbf{X} = \begin{bmatrix} (\Delta\mathbf{x}^{(1)})^\top \\ \vdots \\ (\Delta\mathbf{x}^{(m)})^\top \end{bmatrix} \tag{2.43}$$

$$\Delta\mathbf{f} = \left[f(\mathbf{x} + \Delta\mathbf{x}^{(1)}) - f(\mathbf{x}), \ldots, f(\mathbf{x} + \Delta\mathbf{x}^{(m)}) - f(\mathbf{x}) \right] \tag{2.44}$$

The optimization problem can then be written as

$$\underset{\mathbf{g}}{\text{minimize}} \|\Delta\mathbf{X}\mathbf{g} - \Delta\mathbf{f}\|^2 \tag{2.45}$$

This is a linear regression problem that can be solved exactly by finding the pseudoinverse of $\Delta\mathbf{X}$ and multiplying by $\Delta\mathbf{f}$: [18]

$$\mathbf{g} = \Delta\mathbf{X}^+ \Delta\mathbf{f} \tag{2.46}$$

Algorithm 2.3 provides an implementation of this approach in which the perturbations are drawn uniformly from a hypersphere with radius δ. Figure 2.7 illustrates this approach on a simple two-dimensional function.

[18] Here, $\mathbf{X}^+$ denotes the pseudoinverse of $\mathbf{X}$, which is discussed in more detail in section 13.2. The function `pinv` in Julia computes the pseudoinverse of a matrix. Instead of using the pseudoinverse, we could also use the least-squares solver built into Julia (discussed in appendix A.1.5). Using the *backslash operator*, we can write the solution as `ΔX \ Δf` instead of `pinv(ΔX) * Δf`.

```
function regression_gradient(f, x, m, δ)
    fx = f(x)
    n = length(x)
    ΔX = stack(δ.*normalize(randn(n)) for _ in 1:m)'
    Δf = [f(x + Δx) - fx for Δx in eachrow(ΔX)]
    return ΔX \ Δf
end
```

Algorithm 2.3. A method for estimating the gradient of a function `f` at `x` using finite differences with multiple samples. Perturbation vectors are generated by normalizing `m` normally distributed samples and scaling by a perturbation scalar `δ`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

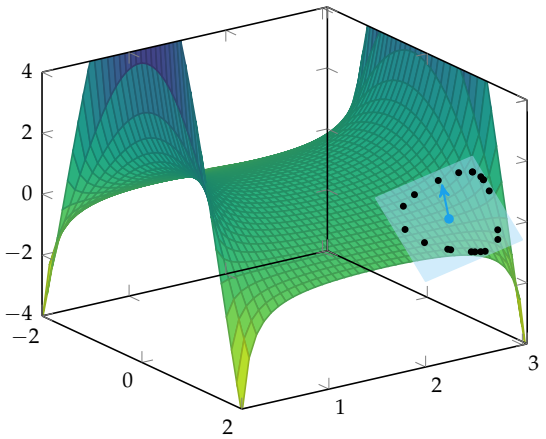


Figure 2.7. Using the regression gradient method to estimate the gradient of the function $f(\mathbf{x}) = \cos(x_1)/\sin(x_2)$, evaluated at $\mathbf{x} = [1.5, 2.5]$ with $m = 20$ samples.

2.6 Simultaneous Perturbation Stochastic Gradient Approximation

The *simultaneous perturbation stochastic gradient approximation* (*SPSA*)[19] can produce a stochastic estimate of the gradient using as few as two function evaluations, which is significantly more efficient than finite differences for large problems. The method is particularly useful when there are a large number of design variables, such as in deep learning applications where there may be billions of parameters and an approximate gradient is sufficient.[20]

This approach approximates the gradient using directional derivatives of f. A single directional derivative is obtained using a central difference with a perturbation $\mathbf{z}$ drawn from a multidimensional Rademacher distribution[21] and a coordinate step size δ:

$$\nabla_{\mathbf{z}} f(\mathbf{x}) \approx \frac{f(\mathbf{x} + \delta\mathbf{z}) - f(\mathbf{x} - \delta\mathbf{z})}{2\delta} \tag{2.47}$$

A single-sample SPSA estimate is this perturbation scaled by this directional derivative:

$$\nabla f(\mathbf{x}) \approx (\nabla_{\mathbf{z}} f(\mathbf{x}))\mathbf{z} \tag{2.48}$$

Averaging many samples can improve the gradient estimate, as done in algorithm 2.4. However, the sample count is typically left quite small or even set to 1 even though the gradient estimate is noisy in the interest of computational efficiency. When used in conjunction with a gradient-based optimization algorithm,

[19] J. C. Spall, "Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation," *IEEE Transactions on Automatic Control*, vol. 37, pp. 332–341, 1992.

[20] See, for example, F. Chaubard and M. J. Kochenderfer, "Scaling Recurrent Neural Networks to a Billion Parameters with Zero-Order Optimization," 2025. arXiv: 2505.17852.

[21] The Rademacher distribution has a 50% chance of producing -1 and a 50% chance of producing 1, which avoids unnecessary probability mass near the origin. J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley, 2003.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

the noisy gradient estimate can still provide a useful direction to search for a solution.

```
function simultaneous_perturbation_gradient(x, f, δ, m)
    n = length(x)
    ∇ = zeros(n)
    for i in 1:m
        z = rand([-1, 1], n)
        ∇ += ((f(x + δ*z) - f(x - δ*z))./(2δ)) * z
    end
    return ∇ ./ m
end
```

Algorithm 2.4. A method for estimating the gradient of a function `f` at `x` by averaging `m` directional derivatives via the simultaneous perturbation gradient approximation. Perturbations are zero-mean with componentwise step size `δ`.

2.7 *Summary*

- Derivatives are useful in optimization because they provide information about how to change a given point in order to improve the objective function.
- For multivariate functions, various derivative-based concepts are useful for directing the search for an optimum, including the gradient, the Hessian, and the directional derivative.
- One approach to numerical differentiation includes finite difference approximations.
- The complex step method can eliminate the effect of subtractive cancellation error when taking small steps, resulting in high-quality gradient estimates.
- Analytic differentiation methods include forward and reverse accumulation on computational graphs.
- The regression gradient uses linear regression to find the best estimate of the gradient with respect to a set of sampled perturbations and their function evaluations.
- Simultaneous perturbation stochastic gradient approximation allows for the estimation of gradients for very large objective functions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

2.8 Exercises

Exercise 2.1. Adopt the forward difference method to approximate the Hessian of $f(\mathbf{x})$ using its gradient, $\nabla f(\mathbf{x})$.

Solution: An entry of the Hessian can be computed using the forward difference method:

$$H_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \approx \frac{\nabla f(\mathbf{x} + h\mathbf{e}_j)_i - \nabla f(\mathbf{x})_i}{h}$$

We can thus approximate the jth column of the Hessian using:

$$\mathbf{H}_{\cdot j} \approx \frac{\nabla f(\mathbf{x} + h\mathbf{e}_j) - \nabla f(x)}{h}$$

where $\mathbf{e}_j$ is the jth basis vector with its jth component equal to one and all other entries equal to zero.

This procedure can be repeated for each column of the Hessian.

Exercise 2.2. What is a drawback of the central difference method over other finite difference methods if we already know $f(\mathbf{x})$?

Solution: It requires two evaluations of the objective function.

Exercise 2.3. Compute the gradient of $f(x) = \ln x + e^x + \frac{1}{x}$ for a point x close to zero. What term dominates in the expression?

Solution: $f'(x) = \frac{1}{x} + e^x - \frac{1}{x^2}$. When x is close to zero, we find that $x < 1$. Hence $\frac{1}{x} > 1$, and finally $\frac{1}{x^2} > \frac{1}{x} > 0$, so $-\frac{1}{x^2}$ dominates.

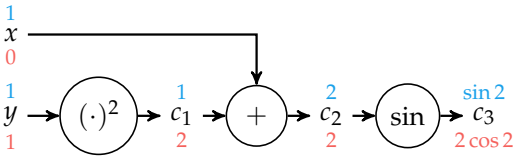
Exercise 2.4. Suppose $f(x)$ is a real-valued function that is also defined for complex inputs. If $f(3 + ih) = 2 + 4ih$, what is $f'(3)$?

Solution: From the complex step method, we have $f'(x) \approx \operatorname{Im}(2 + 4ih)/h = 4h/h = 4$.

Exercise 2.5. Draw the computational graph for $f(x, y) = \sin(x + y^2)$. Use the computational graph with forward accumulation to compute $\partial f/\partial y$ at $(x, y) = (1, 1)$. Label the intermediate values and partial derivatives as they are propagated through the graph.

Solution: See the picture below:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 2.6. Combine the forward and backward difference methods to obtain a difference method for estimating the second-order derivative of a function f at x using three function evaluations.

Solution: The second-order derivative can be approximated using the central difference on the first-order derivative:

$$f''(x) \approx \frac{f'(x + h/2) - f'(x - h/2)}{h}$$

for small values of h.

Substituting in the forward and backward difference estimates of $f'(x + h/2)$ and $f'(x - h/2)$ yields:

$$\begin{aligned} f''(x) &\approx \frac{\frac{f(x+h/2+h/2)-f(x+h/2-h/2)}{h} - \frac{f(x-h/2+h/2)-f(x-h/2-h/2)}{h}}{h} \\ &= \frac{f(x+h) - f(x)}{h^2} - \frac{f(x) - f(x-h)}{h^2} \\ &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \end{aligned}$$

Exercise 2.7. The forward difference derivative approximation evaluates the objective function at x and $x + h$, and divides by h. Computers use floating point numbers with a finite number of bits. As such, adding h to x can produce a floating point number that is not exactly $x + h$. This difference can lead to inaccuracies.

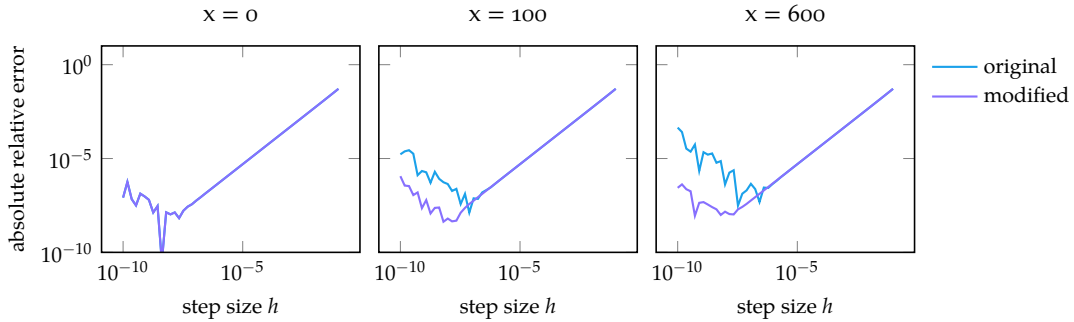
Plot the absolute relative error of the derivative estimate on $f(x) = \exp(x)$ for $x = 0$, $x = 100$, and $x = 600$ for both the forward difference method introduced in the text and a modification that ensures that the step size we divide by is consistent with the difference between x and $x + h$:

$$f'(x) \approx \frac{f(x + h) - f(x)}{(x + h) - x}$$

Does this modification make a difference?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The modification can make a difference for very small step sizes and for large differences in objective function values:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3 Bracketing

This chapter presents a variety of *bracketing* methods for univariate functions, or functions involving a single variable. Bracketing is the process of identifying an interval in which a local minimum lies and then successively shrinking the interval. For many functions, derivative information can be helpful in directing the search for an optimum, but, for some functions, this information may not be available or might not exist. This chapter outlines a wide variety of approaches that leverage different assumptions. Later chapters that consider multivariate optimization will build upon the concepts introduced here.

3.1 Unimodality

Several of the algorithms presented in this chapter assume *unimodality* of the objective function. A *unimodal function* f is one where there is a *unique* x^*, such that f is monotonically decreasing for $x \leq x^*$ and monotonically increasing for $x \geq x^*$. It follows from this definition that the unique global minimum is at x^*, and there are no other local minima.[1] A *multimodal function* is one with multiple peaks and valleys.

Given a unimodal function, we can *bracket* an interval $[a, c]$ containing the global minimum if we can find three points $a < b < c$, such that $f(a) > f(b) < f(c)$. Figure 3.1 shows an example.

[1] It is perhaps more conventional to define unimodal functions in the opposite sense, such that there is a unique global *maximum* rather than a minimum. However, in this text, we try to minimize functions, and so we use the definition in this paragraph.

3.2 Finding an Initial Bracket

When optimizing a function, we often start by first bracketing an interval containing a local minimum. We then successively reduce the size of the bracketed interval to converge on the local minimum. A simple procedure (algorithm 3.1)

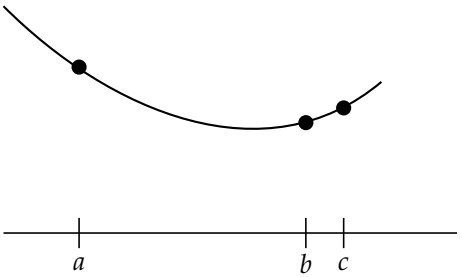


Figure 3.1. Three points shown bracketing a minimum.

can be used to find an initial bracket. Starting at a given point, we take a step in the positive direction. The distance we take is a *hyperparameter* to this algorithm,[2] but the algorithm provided defaults it to 10^{-2}. We then search in the downhill direction to find a new point that exceeds the lowest point. With each step, we expand the step size by some factor, which is another hyperparameter to this algorithm that is often set to 2. An example is shown in figure 3.2. Functions without local minima, such as $\exp(x)$, cannot be bracketed and will cause `bracket_minimum` to fail.

[2] A hyperparameter is a parameter that governs the function of an algorithm. It can be set by an expert or tuned using an optimization algorithm. Many of the algorithms in this text have hyperparameters. We often provide default values suggested in the literature. The success of an algorithm can be sensitive to the choice of hyperparameter.

```
function bracket_minimum(f, x=0; s=1e-2, k=2.0)
    a, ya = x, f(x)
    b, yb = a + s, f(a + s)
    if yb > ya
        a, b = b, a
        ya, yb = yb, ya
        s = -s
    end
    while true
        c, yc = b + s, f(b + s)
        if yc > yb
            return a < c ? (a, c) : (c, a)
        end
        a, ya, b, yb = b, yb, c, yc
        s *= k
    end
end
```

Algorithm 3.1. An algorithm for bracketing an interval in which a local minimum must exist. It takes as input a univariate function `f` and starting position `x`, which defaults to 0. The starting step size `s` and the expansion factor `k` can be specified. It returns a tuple containing the new interval $[a, b]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

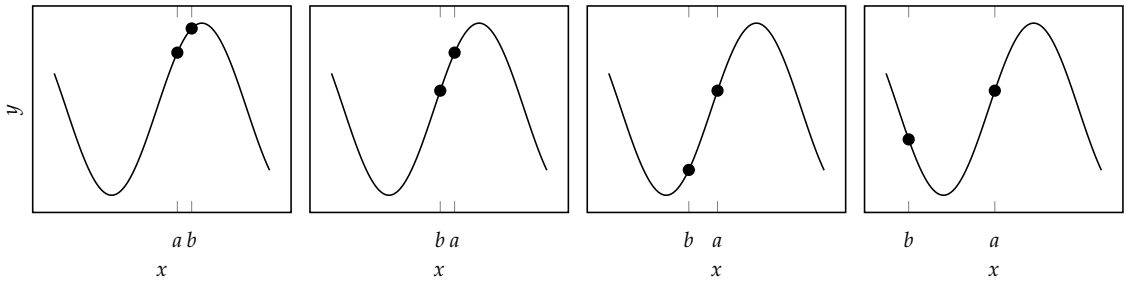


Figure 3.2. An example of running `bracket_minimum` on a function. The method reverses direction between the first and second iteration and then expands until a minimum is bracketed in the fourth iteration.

3.3 Fibonacci Search

Suppose we have a unimodal f bracketed by the interval $[a, b]$. Given a limit on the number of times we can query the objective function, *Fibonacci search* (algorithm 3.2) is guaranteed to maximally shrink the bracketed interval.

Suppose we can query f only twice. If we query f on the one-third and two-third points on the interval, then we are guaranteed to remove one-third of our interval, regardless of f, as shown in figure 3.3.

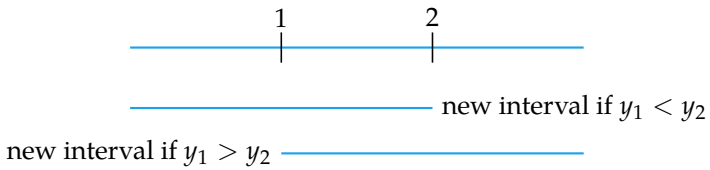


Figure 3.3. Our initial guess for two queries will remove one-third of the initial interval.

We can guarantee a tighter bracket by moving our guesses toward the center. In the limit as $\epsilon \to 0$, we are guaranteed to shrink our interval by a factor of two as shown in figure 3.4.

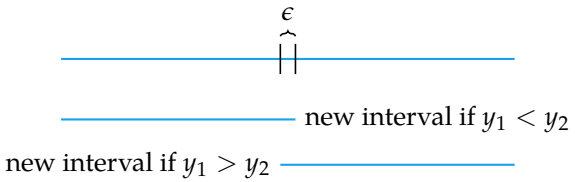


Figure 3.4. The most we can guarantee to shrink our interval is by just under a factor of two.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

With three queries, we can shrink the interval by a factor of three. We first query f on the one-third and two-third points on the interval, discard one-third of the interval, and then sample just next to the better sample as shown in figure 3.5.

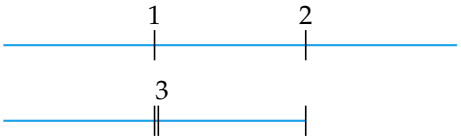


Figure 3.5. With three queries we can shrink the domain by a factor of three. The third query is made based on the result of the first two queries.

For n queries, the interval lengths are related to the Fibonacci sequence: 1, 1, 2, 3, 5, 8, and so forth. The first two terms are one, and the following terms are always the sum of the previous two:

$$F_n = \begin{cases} 1 & \text{if } n \leq 2 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases} \tag{3.1}$$

Figure 3.6 shows the relationship between the intervals. Example 3.1 walks through an application to a univariate function.

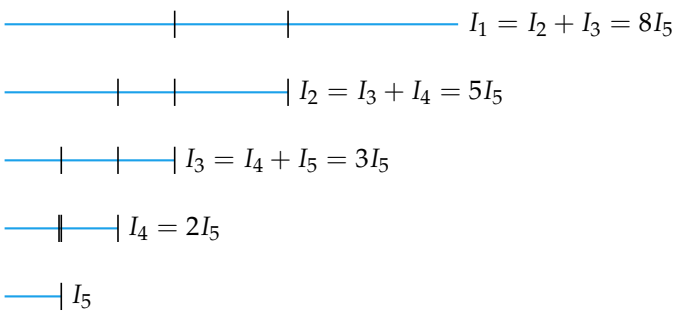


Figure 3.6. For n queries we are guaranteed to shrink our interval by a factor of F_{n+1}. The length of every interval constructed during Fibonacci search can be expressed in terms of the final interval times a Fibonacci number. If the final, smallest interval has length I_n, then the second smallest interval has length $I_{n-1} = F_3 I_n$, the third smallest interval has length $I_{n-2} = F_4 I_n$, and so forth.

The Fibonacci sequence can be determined analytically using *Binet's formula*:

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}, \tag{3.2}$$

where $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$ is the *golden ratio*.

The ratio between successive values in the Fibonacci sequence is:

$$\frac{F_n}{F_{n-1}} = \varphi \frac{1 - s^n}{1 - s^{n-1}} \tag{3.3}$$

where $s = (1 - \sqrt{5})/(1 + \sqrt{5}) \approx -0.382$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function fibonacci_search(f, a, b, n; ϵ=0.01)
    s = (1-√5)/(1+√5)
    ρ = 1 / (φ*(1-s^(n+1))/(1-s^n))
    r = (1-ρ)*a + ρ*b # right sample point
    yr, n = f(r), n-1
    while n > 0
        if n > 1
            l = ρ*a + (1-ρ)*b
        else
            l = ϵ*a + (1-ϵ)*r
        end
        ρ = 1 / (φ*(1-s^(n+1))/(1-s^n))
        yl, n = f(l), n-1
        if yl < yr
            r, b, yr = l, r, yl
        else
            a, b = b, l
        end
    end
    return a < b ? (a, b) : (b, a)
end
```

Algorithm 3.2. Fibonacci search to be run on univariate function `f`, with bracketing interval $[a, b]$, for `n` > 1 function evaluations. It returns the new interval `(a, b)`. The optional parameter `ϵ` controls the lowest-level interval. The golden ratio `φ` is defined in `Base.MathConstants.jl`.

3.4 Golden Section Search

If we take the limit for large n, we see that the ratio between successive values of the Fibonacci sequence approaches the golden ratio:

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \varphi \tag{3.4}$$

Golden section search (algorithm 3.3) uses the golden ratio to approximate Fibonacci search. Figure 3.7 shows the relationship between the intervals. Figures 3.8 and 3.9 compare Fibonacci search with golden section search on unimodal and non-unimodal functions, respectively.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 3.1. Using Fibonacci search with five function evaluations to optimize a univariate function.

Consider using Fibonacci search with five function evaluations to minimize $f(x) = \exp(x-2) - x$ over the interval $[a, b] = [-2, 6]$. The first two function evaluations are made at $\frac{F_5}{F_6}$ and $1 - \frac{F_5}{F_6}$, along the length of the initial bracketing interval:

$$f(x^{(1)}) = f\left(a + (b-a)\left(1 - \frac{F_5}{F_6}\right)\right) = f(1) = -0.632$$
$$f(x^{(2)}) = f\left(a + (b-a)\frac{F_5}{F_6}\right) = f(3) = -0.282$$

The evaluation at $x^{(1)}$ is lower, yielding the new interval $[a, b] = [-2, 3]$. Two evaluations are needed for the next interval split:

$$x_{\text{left}} = a + (b-a)\left(1 - \frac{F_4}{F_5}\right) = 0$$
$$x_{\text{right}} = a + (b-a)\frac{F_4}{F_5} = 1$$

A third function evaluation is thus made at x_{left}, as x_{right} has already been evaluated:

$$f(x^{(3)}) = f(0) = 0.135$$

The evaluation at $x^{(1)}$ is lower, yielding the new interval $[a, b] = [0, 3]$. Two evaluations are needed for the next interval split:

$$x_{\text{left}} = a + (b-a)\left(1 - \frac{F_3}{F_4}\right) = 1$$
$$x_{\text{right}} = a + (b-a)\frac{F_3}{F_4} = 2$$

A fourth function evaluation is thus made at x_{right}, as x_{left} has already been evaluated:

$$f(x^{(4)}) = f(2) = -1$$

The new interval is $[a, b] = [1, 3]$. A final evaluation is made just next to the center of the interval at $2 + \epsilon$, and it is found to have a slightly higher value than $f(2)$. The final interval is $[1, 2 + \epsilon]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

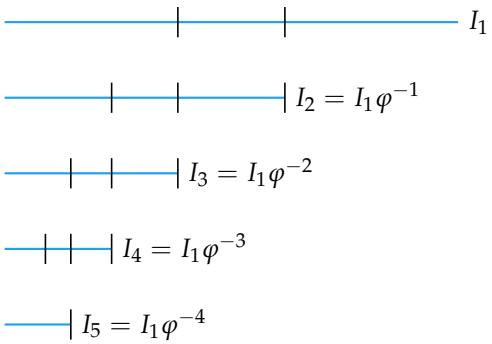


Figure 3.7. For n queries of a univariate function we are guaranteed to shrink a bracketing interval by a factor of φ^{n-1}.

```
function golden_section_search(f, a, b, n)
    ρ = φ-1
    d = ρ * b + (1 - ρ)*a
    yd = f(d)
    for i = 1 : n-1
        c = ρ*a + (1 - ρ)*b
        yc = f(c)
        if yc < yd
            b, d, yd = d, c, yc
        else
            a, b = b, c
        end
    end
    return a < b ? (a, b) : (b, a)
end
```

Algorithm 3.3. Golden section search to be run on a univariate function `f`, with bracketing interval $[a, b]$, for `n` > 1 function evaluations. It returns the new interval (`a`, `b`). Julia already has the golden ratio `φ` defined. Guaranteeing convergence to within ϵ requires $n = \lceil \ln((b-a)/\epsilon) / \ln \varphi \rceil$ iterations.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

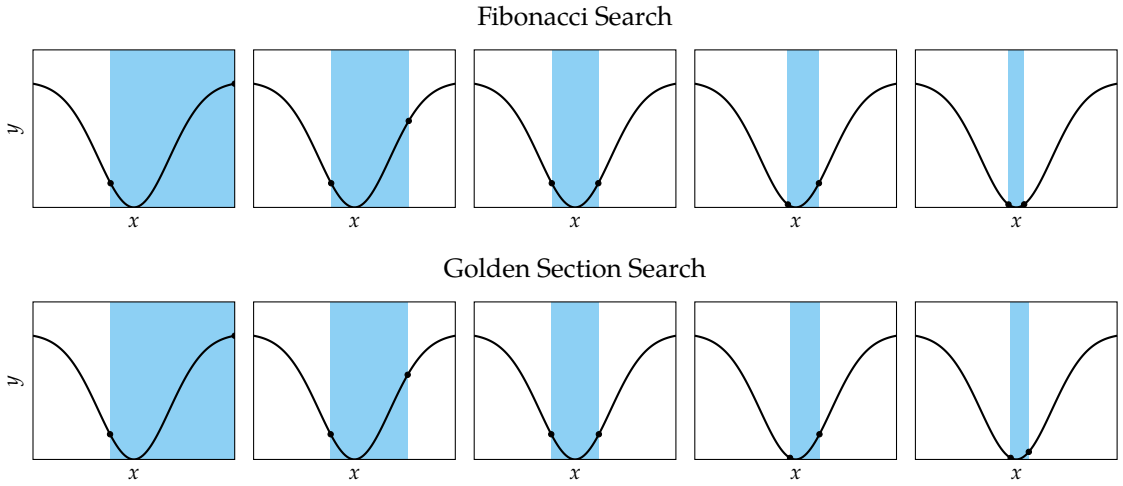


Figure 3.8. Fibonacci and golden section search on a unimodal function.

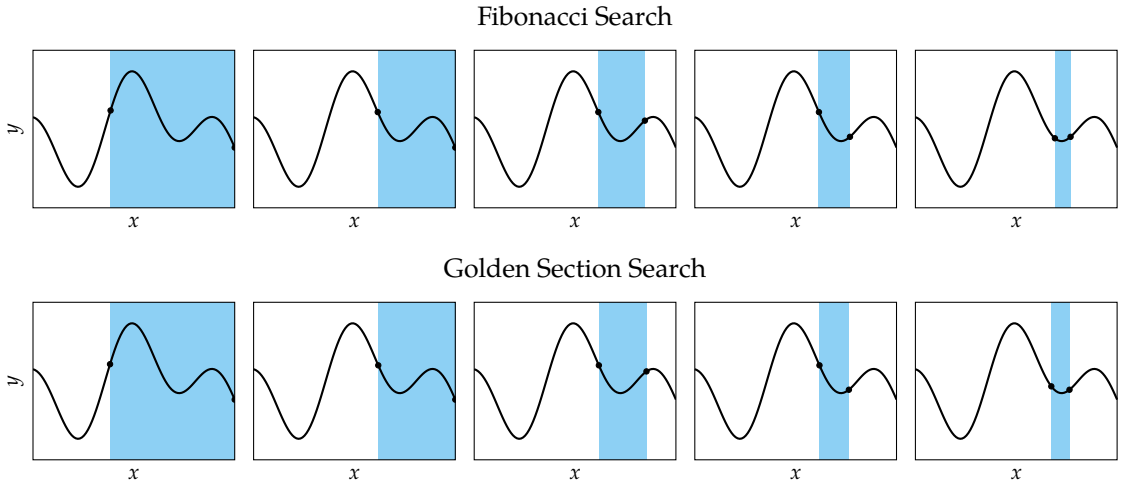


Figure 3.9. Fibonacci and golden section search on a nonunimodal function. Search is not guaranteed to find a global minimum.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3.5 Quadratic Fit Search

Quadratic fit search leverages our ability to analytically solve for the minimum of a quadratic function. Many local minima look quadratic when we zoom in close enough. Quadratic fit search iteratively fits a quadratic function to three bracketing points, solves for the minimum, chooses a new set of bracketing points, and repeats as shown in figure 3.10.

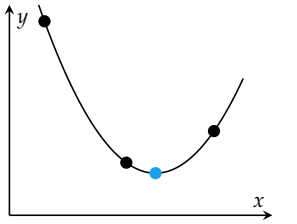


Figure 3.10. Quadratic fit search fits a quadratic function to three bracketing points (black dots) and uses the analytic minimum (blue dot) to determine the next set of bracketing points.

Given bracketing points $a < b < c$, we wish to find the coefficients p_1, p_2, and p_3 for the quadratic function q that goes through (a, y_a), (b, y_b), and (c, y_c):

$$q(x) = p_1 + p_2 x + p_3 x^2 \tag{3.5}$$

$$y_a = p_1 + p_2 a + p_3 a^2 \tag{3.6}$$

$$y_b = p_1 + p_2 b + p_3 b^2 \tag{3.7}$$

$$y_c = p_1 + p_2 c + p_3 c^2 \tag{3.8}$$

In matrix form, we have

$$\begin{bmatrix} y_a \\ y_b \\ y_c \end{bmatrix} = \begin{bmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \tag{3.9}$$

We can solve for the coefficients through matrix inversion:

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 1 & a & a^2 \\ 1 & b & b^2 \\ 1 & c & c^2 \end{bmatrix}^{-1} \begin{bmatrix} y_a \\ y_b \\ y_c \end{bmatrix} \tag{3.10}$$

Our quadratic function is then

$$q(x) = y_a \frac{(x-b)(x-c)}{(a-b)(a-c)} + y_b \frac{(x-a)(x-c)}{(b-a)(b-c)} + y_c \frac{(x-a)(x-b)}{(c-a)(c-b)} \tag{3.11}$$

We can solve for the unique minimum by finding where the derivative is zero:

$$x^* = \frac{1}{2} \frac{y_a(b^2 - c^2) + y_b(c^2 - a^2) + y_c(a^2 - b^2)}{y_a(b - c) + y_b(c - a) + y_c(a - b)} \tag{3.12}$$

Quadratic fit search is typically faster than golden section search. It may need safeguards for cases where the next point is very close to other points. A basic implementation is provided in algorithm 3.4. Figure 3.11 shows several iterations of the algorithm.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function quadratic_fit_search(f, a, b, c, n)
    ya, yb, yc = f(a), f(b), f(c)
    for i in 1:n-3
        x = 0.5*(ya*(b^2-c^2)+yb*(c^2-a^2)+yc*(a^2-b^2)) /
                (ya*(b-c)    +yb*(c-a)    +yc*(a-b))
        yx = f(x)
        if x > b
            if yx > yb
                c, yc = x, yx
            else
                a, ya, b, yb = b, yb, x, yx
            end
        elseif x < b
            if yx > yb
                a, ya = x, yx
            else
                c, yc, b, yb = b, yb, x, yx
            end
        end
    end
    return (a, b, c)
end
```

Algorithm 3.4. Quadratic fit search to be run on univariate function `f`, with bracketing interval $[a, c]$ with `a` $<$ `b` $<$ `c`. The method will run for `n` function evaluations. It returns the new bracketing values as a tuple, `(a, b, c)`.

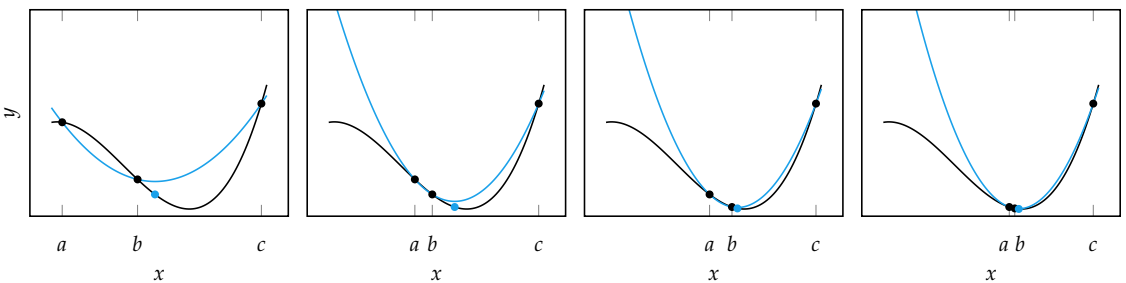


Figure 3.11. Four iterations of the quadratic fit method.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3.6 Shubert-Piyavskii Method

In contrast with previous methods in this chapter, the *Shubert-Piyavskii method*[3] is a *global optimization method* over a domain $[a, b]$, meaning it is guaranteed to converge on the global minimum of a function irrespective of any local minima or whether the function is unimodal. A basic implementation is provided by algorithm 3.5.

The Shubert-Piyavskii method requires that the function be *Lipschitz continuous*,[4] meaning that it is continuous and there is an upper bound on the magnitude of its derivative. A function f is Lipschitz continuous on $[a, b]$ if there exists an $\ell > 0$ such that:[5]

$$|f(x) - f(y)| \leq \ell|x - y| \text{ for all } x, y \in [a, b] \tag{3.13}$$

Intuitively, ℓ is as large as the largest unsigned instantaneous rate of change the function attains on $[a, b]$. Given a point $(x_0, f(x_0))$, we know that the lines $f(x_0) - \ell(x - x_0)$ for $x > x_0$ and $f(x_0) + \ell(x - x_0)$ for $x < x_0$ form a lower bound of f.

The Shubert-Piyavskii method iteratively builds a tighter and tighter lower bound on the function. Given a valid Lipschitz constant ℓ, the algorithm begins by sampling the midpoint, $x^{(1)} = (a + b)/2$. A sawtooth lower bound is constructed using lines of slope $\pm\ell$ from this point. These lines will always lie below f if ℓ is a valid Lipschitz constant as shown in figure 3.12.

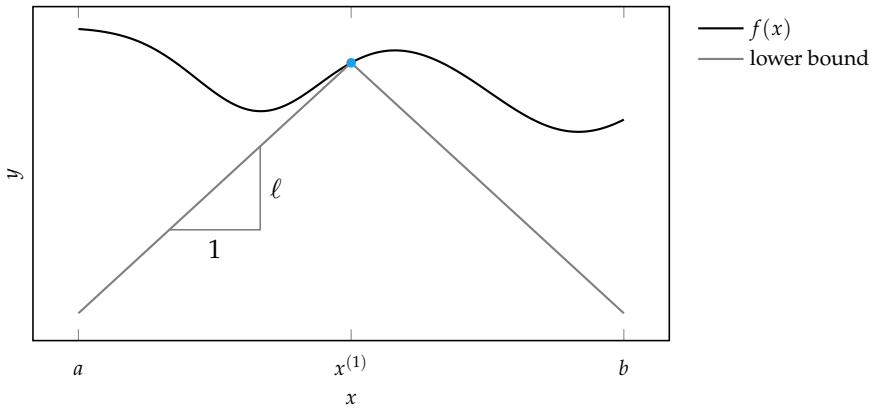


Figure 3.12. The first iteration of the Shubert-Piyavskii method.

[3] S. Piyavskii, "An Algorithm for Finding the Absolute Extremum of a Function," *USSR Computational Mathematics and Mathematical Physics*, vol. 12, no. 4, pp. 57–67, 1972. B. O. Shubert, "A Sequential Method Seeking the Global Maximum of a Function," *SIAM Journal on Numerical Analysis*, vol. 9, no. 3, pp. 379–388, 1972.

[4] This property is named for the German mathematician Rudolf Lipschitz (1832–1903).

[5] We can extend the definition of Lipschitz continuity to multivariate functions, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and the absolute value is replaced by any vector norm.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Upper vertices in the sawtooth correspond to sampled points. Lower vertices correspond to intersections between the Lipschitz lines originating from each sampled point. Further iterations find the minimum point in the sawtooth, evaluate the function at that x value, and then use the result to update the sawtooth. Figure 3.13 illustrates this process.

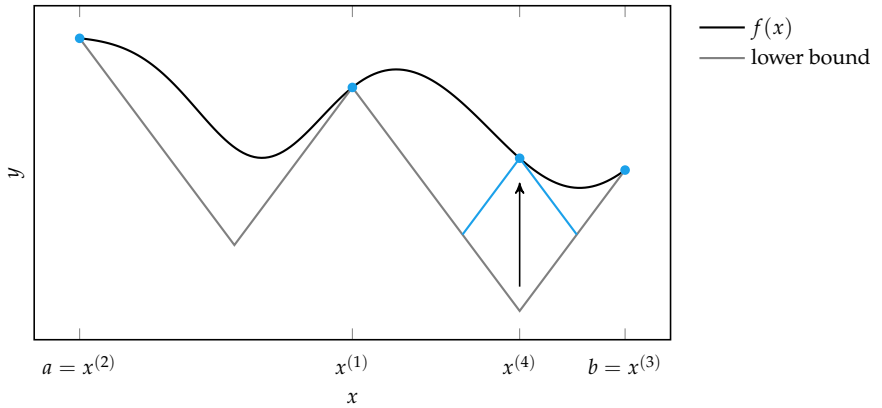


Figure 3.13. Updating the lower bound involves sampling a new point and intersecting the new lines with the existing sawtooth.

The algorithm is typically stopped when the difference in height between the minimum sawtooth value and the function evaluation at that point is less than a given tolerance ϵ. For the minimum peak $(x^{(n)}, y^{(n)})$ and function evaluation $f(x^{(n)})$, we thus terminate if $f(x^{(n)}) - y^{(n)} < \epsilon$.

The regions in which the minimum could lie can be computed using this update information. For every peak, an uncertainty region can be computed according to:

$$\left[x^{(i)} - \frac{1}{\ell}(y_{\text{min}} - y^{(i)}), x^{(i)} + \frac{1}{\ell}(y_{\text{min}} - y^{(i)})\right] \tag{3.14}$$

for each sawtooth lower vertex $(x^{(i)}, y^{(i)})$ and the minimum sawtooth upper vertex $(x_{\text{min}}, y_{\text{min}})$. A point will contribute an uncertainty region only if $y^{(i)} < y_{\text{min}}$. The minimum is located in one of these peak uncertainty regions.

The main drawback of the Shubert-Piyavskii method is that it requires knowing a valid Lipschitz constant. Large Lipschitz constants will result in poor lower bounds. Figure 3.14 shows several iterations of the Shubert-Piyavskii method.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function shubert_piyavskii(f, a, b, l, ϵ)
    x1 = (a+b)/2
    pts = [(x=x1, y=f(x1))]

    Δ = Inf
    while Δ > ϵ
        best = (i=0, x=0.0, y=Inf)
        # consider leftmost point
        y = pts[1].y - l*(pts[1].x-a)
        if y < best.y
            best = (i=1, x=a, y=y)
        end
        # consider rightmost point
        y = pts[end].y - l*(b-pts[end].x)
        if y < best.y
            best = (i=length(pts)+1, x=b, y=y)
        end
        # consider interior points
        for i in 2:length(pts)
            A, B = pts[i-1], pts[i]
            t = ((A.y - B.y) - l*(A.x - B.x)) / (2l)
            P = (x=A.x + t, y=A.y - t*l)
            if P.y < best.y
                best = (i=i, x=P.x, y=P.y)
            end
        end
        insert!(pts, best.i, (x=best.x, y=f(best.x)))
        Δ = pts[best.i].y - best.y
    end
    return pts[argmin(P.y for P in pts)]
end
```

Algorithm 3.5. The Shubert-Piyavskii method to be run on univariate function `f`, with bracketing interval `a` < `b` and Lipschitz constant `l`. The algorithm runs until the update is less than the tolerance `ϵ`. The method returns the best point, which includes its `x` location and `y` value.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

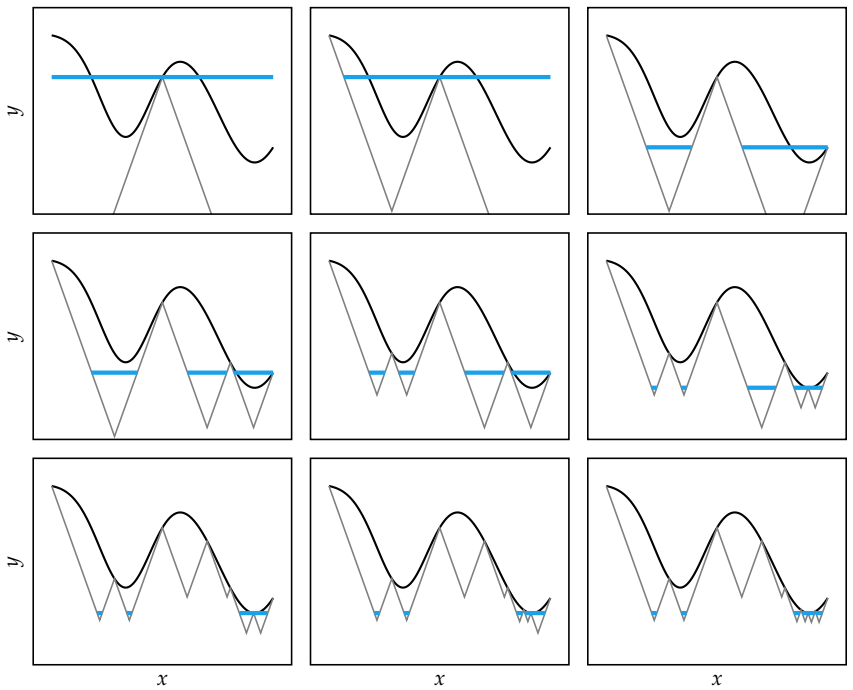


Figure 3.14. Nine iterations of the Shubert-Piyavskii method proceeding left to right and top to bottom. The blue lines are uncertainty regions in which the global minimum could lie.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3.7 Bisection Method

The *bisection method* (algorithm 3.6) can be used to find *roots* of a function, or points where the function is zero. Such *root-finding methods* can be used for optimization by applying them to the derivative of the objective, locating where $f'(x) = 0$. In general, we must ensure that the resulting points are indeed local minima.

The bisection method maintains a bracket $[a, b]$ in which at least one root is known to exist. If f is continuous on $[a, b]$, and there is some $y \in [f(a), f(b)]$, then the *intermediate value theorem* stipulates that there exists at least one $x \in [a, b]$, such that $f(x) = y$ as shown in figure 3.15. It follows that a bracket $[a, b]$ is guaranteed to contain a zero if $f(a)$ and $f(b)$ have opposite signs.

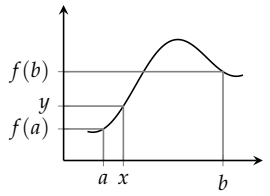


Figure 3.15. A horizontal line drawn from any $y \in [f(a), f(b)]$ must intersect the graph at least once.

The bisection method cuts the bracketed region in half with every iteration. The midpoint $(a + b)/2$ is evaluated, and the new bracket is formed from the midpoint and whichever side continues to bracket a zero. We can terminate immediately if the midpoint evaluates to zero. Otherwise we can terminate after a fixed number of iterations. Figure 3.16 shows four iterations of the bisection method. This method is guaranteed to converge within ϵ of x^* within $\lg\left(\frac{|b-a|}{\epsilon}\right)$ iterations, where lg denotes the base 2 logarithm.

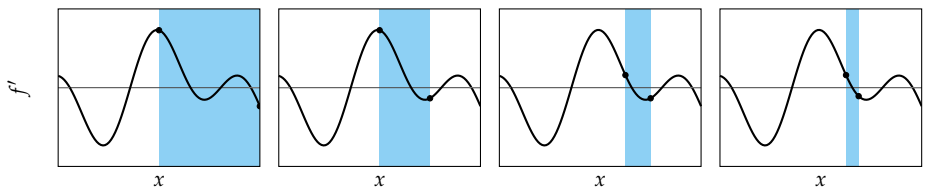


Figure 3.16. Four iterations of the bisection method. The horizontal line corresponds to $f'(x) = 0$. Note that multiple roots exist within the initial bracket.

Root-finding algorithms like the bisection method require starting intervals $[a, b]$ on opposite sides of a zero. That is, $\text{sign}(f'(a)) \neq \text{sign}(f'(b))$, or equivalently, $f'(a)f'(b) \leq 0$. Algorithm 3.7 provides a method for automatically determining such an interval. It starts with a guess interval $[a, b]$. So long as the interval is invalid, its width is increased by a constant factor. Doubling the interval size is a common choice. This method will not always succeed as shown in figure 3.17. Functions that have two nearby roots can be missed, causing the interval to infinitely increase without termination.

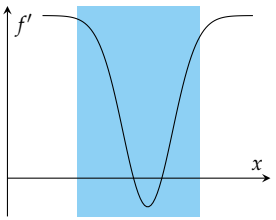


Figure 3.17. A bracketing method initialized such that it straddles the two roots in this figure will expand forever, never to find a sign change. Also, if the initial interval is between the two roots, doubling the interval can cause both ends of the interval to simultaneously pass the two roots.

The *Brent-Dekker* method is an extension of the bisection method. It is a root-finding algorithm that combines elements of the secant method (section 6.2) and inverse quadratic interpolation. It has reliable and fast convergence properties, and

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function bisection(f′, a, b, ϵ)
    if a > b; a,b = b,a; end # ensure a < b

    ya, yb = f′(a), f′(b)
    if ya == 0; b = a; end
    if yb == 0; a = b; end
    sign_ya = sign(ya)

    while b - a > ϵ
        x = (a+b)/2
        y = f′(x)
        if y == 0
            a, b = x, x
        elseif sign(y) == sign_ya
            a = x
        else
            b = x
        end
    end

    return (a,b)
end
```

Algorithm 3.6. The bisection algorithm, where `f′` is the derivative of the univariate function we seek to optimize. We have `a` $<$ `b` that bracket a zero of `f′`. The interval width tolerance is `ϵ`. Calling `bisection` returns the new bracketed interval $[a, b]$ as a tuple.

The prime character `′` is not an apostrophe. Thus, `f′` is a variable name rather than a transposed vector `f`. The symbol can be created by typing `\prime` and hitting tab.

```
function bracket_sign_change(f′, a, b; k=2)
    if a > b; a,b = b,a; end # ensure a < b

    center, half_width = (b+a)/2, (b-a)/2
    while f′(a)*f′(b) > 0
        half_width *= k
        a = center - half_width
        b = center + half_width
    end

    return (a,b)
end
```

Algorithm 3.7. An algorithm for finding an interval in which a sign change occurs. The inputs are the real-valued function `f′` defined on the real numbers, and starting interval $[a, b]$. It returns the new interval as a tuple by expanding the interval width until there is a sign change between the function evaluated at the interval bounds. The expansion factor `k` defaults to 2.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

it is the univariate optimization algorithm of choice in many popular numerical optimization packages.[6]

[6] The details of this algorithm can be found in R. P. Brent, *Algorithms for Minimization Without Derivatives*. Prentice Hall, 1973. The algorithm is an extension of the work by T. J. Dekker, "Finding a Zero by Means of Successive Linear Interpolation," in *Constructive Aspects of the Fundamental Theorem of Algebra*, B. Dejon and P. Henrici, eds., Interscience, 1969.

3.8 Summary

- Many optimization methods shrink a bracketing interval, including Fibonacci search, golden section search, and quadratic fit search.
- The Shubert-Piyavskii method outputs a set of bracketed intervals containing the global minima, given the Lipschitz constant.
- Root-finding methods like the bisection method can be used to find where the derivative of a function is zero.

3.9 Exercises

Exercise 3.1. Give an example of a problem when Fibonacci search is preferred over the bisection method.

Solution: Fibonacci search is preferred when derivatives are not available.

Exercise 3.2. What is a drawback of the Shubert-Piyavskii method?

Solution: The Shubert-Piyavskii method needs the Lipschitz constant, which may not be known.

Exercise 3.3. Give an example of a nontrivial function where quadratic fit search would identify the minimum correctly once the function values at three distinct points are available.

Solution: $f(x) = x^2$. Since the function is quadratic, after three evaluations, the quadratic model will represent this function exactly.

Exercise 3.4. Suppose we have $f(x) = x^2/2 - x$. Apply the bisection method to find an interval containing the minimizer of f starting with the interval $[0, 1000]$. Execute three steps of the algorithm.

Solution: We can use the bisection method to find the roots of $f'(x) = x - 1$. After the first update, we have $[0, 500]$. Then, $[0, 250]$. Finally, $[0, 125]$.

Exercise 3.5. Suppose we have a function $f(x) = (x + 2)^2$ on the interval $[0, 1]$. Is 2 a valid Lipschitz constant for f on that interval?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: No, the Lipschitz constant must bound the derivative everywhere on the interval, and $f'(1) = 2(1+2) = 6$.

Exercise 3.6. Suppose we have a unimodal function defined on the interval $[1, 32]$. After three function evaluations of our choice, will we be able to narrow the optimum to an interval of at most length 10? Why or why not?

Solution: No. The best you can do is use Fibonacci Search and shrink the uncertainty by a factor of 3; that is, to $(32-1)/3 = 10\frac{1}{3}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

4 Local Descent

Up to this point, we have focused on optimization involving a single design variable. This chapter introduces a general approach to optimization involving *multivariate* functions, or functions with more than one variable. The focus of this chapter is on how to use *local models* to incrementally improve a design point until some convergence criterion is met. We begin by discussing methods that, at each iteration, choose a descent direction based on a local model and then choose a step size. We then discuss methods that restrict the step to be within a region where the local model is believed to be valid. This chapter concludes with a discussion of convergence conditions. The next two chapters will discuss how to use first- and second-order models built from gradient or Hessian information.

4.1 Descent Direction Iteration

A common approach to optimization is to incrementally improve a design point $\mathbf{x}$ by taking a step that attempts to minimize the objective value based on a local model. The local model may be obtained, for example, from a first- or second-order Taylor approximation. Optimization algorithms that follow this general approach are referred to as *descent direction methods*. They start with a design point $\mathbf{x}^{(1)}$ and then generate a sequence of points, sometimes called *iterates*, to converge to a local minimum.[1]

The iterative descent direction procedure involves the following steps:

1. Check whether $\mathbf{x}^{(k)}$ satisfies the termination conditions. If it does, terminate; otherwise proceed to the next step.
2. Determine the *descent direction* $\mathbf{d}^{(k)}$ using local information such as the gradient or Hessian. A strict definition of descent direction requires $\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) < 0$,

[1] The choice of $\mathbf{x}^{(1)}$ can affect the success of the algorithm in finding a minimum. Domain knowledge is often used to choose a reasonable value. When that is not available, we can search over the design space using the techniques that will be covered in chapter 16. In some cases, we might have a solution from a previous optimization run on either the same or a related problem that we can use as a starting point. This prior solution can be used to *warm start* the optimization algorithm.

which guarantees that moving an infinitesimal amount in direction $\mathbf{d}^{(k)}$ will decrease the objective. However, we will be discussing more general methods, such as momentum, where the descent direction they choose may not strictly satisfy this condition. Some algorithms assume $\|\mathbf{d}^{(k)}\| = 1$, but others do not.

3. Determine the *step factor* $\alpha^{(k)}$. A step factor is sometimes referred to as a *learning rate*, especially in the context of machine learning applications.[2] Some algorithms attempt to optimize the step factor to maximally decrease f.

4. Compute the next design point according to:

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)} \tag{4.1}$$

[2] T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning*, 2nd ed. Springer, 2017.

There are many different optimization methods, each with their own ways of determining α and $\mathbf{d}$. They typically follow a similar iterative structure, as implemented in algorithm 4.1. Some methods that we will discuss when applied to general objective functions may not guarantee descent at every iteration, but they still generally follow this structure.

```
abstract type DescentMethod end

function iterated_descent(M::DescentMethod, f, ∇f, x, k_max)
    init!(M, f, ∇f, x)
    for k in 1:k_max
        x = step!(M, f, ∇f, x)
    end
    return x
end
```

Algorithm 4.1. An iterated descent method for minimizing `f` starting at design `x` using information from the gradient `∇f`. Other methods might additionally use the Hessian, `H`. This implementation operates on an abstract `DescentMethod` object that should support both an `init!` call for initialization and a `step!` implementation for executing a single descent step. It executes `k_max` iterations. Other termination conditions are presented in section 4.6.

4.2 Step Factors

The step factor $\alpha^{(k)}$ influences the size of the step taken in the descent direction. The *step size* at iteration k is the distance from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$. If $\|\mathbf{d}^{(k)}\| = 1$, then the step size is the same as the step factor. Otherwise, the step size is $\alpha^{(k)}\|\mathbf{d}^{(k)}\|$.

Some algorithms use a fixed step factor. Large steps will tend to result in faster convergence but risk overshooting the minimum. Smaller steps tend to be more stable but can result in slower convergence.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Alternatively, we can decay the step factor over time, with the intuition that larger steps are taken early in the optimization process to quickly arrive in the general proximity of a solution and smaller steps are later taken to refine the solution. One common decay scheme is to multiply the step factor by a constant γ at each iteration, where $0 < \gamma < 1$:

$$\alpha^{(k)} = \gamma^k \alpha^{(0)} = \gamma \alpha^{(k-1)} \tag{4.2}$$

This scheme ensures that the step factor decreases over time, which can help descent algorithms converge to a minimum.[3] Smaller values for γ result in faster decay of the step factor, resulting in a faster shift from large steps in design space to smaller, fine-tuning steps.

[3] The convergence of various descent algorithms is discussed in detail by J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. Springer, 2006.

This decay scheme causes $\alpha^{(k)}$ to decrease exponentially. If the magnitude of the descent vector is bounded, $\|\mathbf{d}^{(k)}\| \leq \delta$ for all k, then the total distance a descent method can travel over an infinite number of steps is bounded by

$$\sum_{k=0}^{\infty} \delta \alpha^{(0)} \gamma^k = \frac{\delta \alpha^{(0)}}{1 - \gamma} \tag{4.3}$$

An initial design can thus be sufficiently far from any local minimizer that the iteration stalls before reaching a minimum. Smaller values of γ decay faster but increase that risk.

Step sizes can be decayed without forcing a finite total travel distance. Polynomial decay uses

$$\alpha^{(k)} = \frac{\alpha^{(0)}}{(1 + \beta k)^p} \tag{4.4}$$

for a rate $\beta > 0$ and a power $p > 0$. The total travel distance is unbounded when $0 < p \leq 1$.

A simple alternative is to use exponential decay, but prevent decay below a lower bound $\alpha_{\min} > 0$:

$$\alpha^{(k)} = \max\left(\alpha_{\min}, \gamma^k \alpha^{(0)}\right) \tag{4.5}$$

This ensures fast initial decay while preventing premature stagnation.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

4.3 Line Search

Instead of using a fixed or decaying step factor, we can use *line search* to directly optimize the step factor to minimize the objective given a direction $\mathbf{d}$:

$$\underset{\alpha}{\text{minimize}}\, f(\mathbf{x} + \alpha\mathbf{d}) \tag{4.6}$$

Line search is a univariate optimization problem (chapter 3).[4] To inform the search, we can use the derivative of the line search objective, which is simply the directional derivative along $\mathbf{d}$ at $\mathbf{x} + \alpha\mathbf{d}$. Line search generally searches along the full line, allowing α to be positive or negative. In cases where the search direction is known to lead to improvement, we may only consider positive values for α. Line search is demonstrated in example 4.1 and implemented in algorithm 4.2.

[4] The Brent-Dekker method, mentioned in the previous chapter, is a commonly used univariate optimization method.

```
function line_search(f, x, d)
    objective = α -> f(x + α*d)
    a, b = bracket_minimum(objective)
    α = minimize(objective, a, b)
    return x + α*d
end
```

Algorithm 4.2. A method for conducting a line search, which finds the optimal step factor along a descent direction `d` from design point `x` to minimize function `f`. The `minimize` function can be implemented using a univariate optimization algorithm such as the Brent-Dekker method.

One disadvantage of conducting a line search at each step is the computational cost of optimizing α to a high degree of precision. Instead, it is common to quickly find a reasonable value and then move on, selecting $\mathbf{x}^{(k+1)}$, and then picking a new direction $\mathbf{d}^{(k+1)}$.

Consider conducting a line search on $f(x_1, x_2, x_3) = \sin(x_1 x_2) + \exp(x_2 + x_3) - x_3$ from $\mathbf{x} = [1, 2, 3]$ in the direction $\mathbf{d} = [0, -1, -1]$. The corresponding optimization problem is:

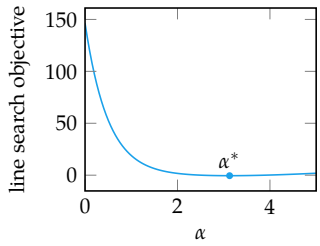
$$\underset{\alpha}{\text{minimize}}\, \sin((1 + 0\alpha)(2 - \alpha)) + \exp((2 - \alpha) + (3 - \alpha)) - (3 - \alpha)$$

which simplifies to:

$$\underset{\alpha}{\text{minimize}}\, \sin(2 - \alpha) + \exp(5 - 2\alpha) + \alpha - 3$$

The minimum is at $\alpha \approx 3.127$ with $\mathbf{x} \approx [1, -1.126, -0.126]$.

Example 4.1. Line search used to minimize a function along a descent direction.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

4.4 Approximate Line Search

It is often more computationally efficient to perform more iterations of a descent method than to do exact line search at each iteration, especially if the function and derivative calculations are expensive. Many of the methods discussed so far can benefit from using *approximate line search* to find a suitable step size with a small number of evaluations. This section discusses methods for searching for a step size that satisfies certain criteria known as the *Wolfe conditions*.[5]

[5] These conditions are named for the American mathematician Philip Wolfe (1927–2016). P. Wolfe, "Convergence Conditions for Ascent Methods," *SIAM Review*, vol. 11, no. 2, pp. 226–235, 1969. P. Wolfe, "Convergence Conditions for Ascent Methods. II: Some Corrections," *SIAM Review*, vol. 13, no. 2, pp. 185–188, 1971.

4.4.1 Sufficient Decrease

The first Wolfe condition requires that the step factor α cause a *sufficient decrease* in the objective function value.[6] If the descent direction is $\mathbf{d}^{(k)}$ at step k, then a first-order approximation of the objective function at $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}$ is given by:

$$f(\mathbf{x}^{(k+1)}) = f(\mathbf{x}^{(k)}) + \alpha\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) \tag{4.7}$$

[6] This condition is sometimes referred to as the *Armijo condition*.

If we have a valid descent direction, the directional derivative $\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)})$ is negative, resulting in a decrease in the objective function value with a sufficiently small step.

Although we could choose a step factor that causes any decrease in the objective function value, it is often beneficial to require a certain amount of decrease related to what could be expected by a first-order approximation. We can define this amount as a fraction $\beta \in [0, 1)$ of the decrease predicted by the first-order approximation,[7] resulting in the condition:

[7] A typical value for β is 10^{-4}.

$$f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(k)}) + \beta\alpha\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) \tag{4.8}$$

Figure 4.1 illustrates this condition. If $\beta = 0$, then any decrease is acceptable. If $\beta = 1$, then the decrease has to be at least as much as what would be predicted by a first-order approximation, but the line search could fail to find a step factor that satisfies the sufficient decrease criterion.[8]

[8] This is demonstrated on an example in exercise 4.3.

To find an α that meets this condition, we can start with a large value and decrease it by a constant factor until the sufficient decrease condition is satisfied. This algorithm is known as *backtracking line search*[9] because of how it backtracks along the descent direction. Backtracking line search is shown in figure 4.2 and implemented in algorithm 4.3. We walk through the procedure in example 4.2.

[9] Also known as *Armijo line search*. L. Armijo, "Minimization of Functions Having Lipschitz Continuous First Partial Derivatives," *Pacific Journal of Mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

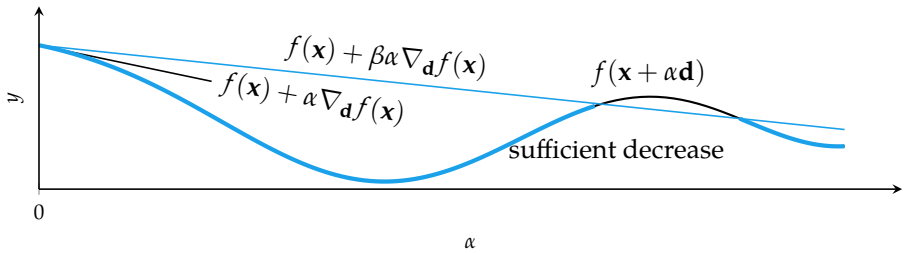


Figure 4.1. The sufficient decrease condition, the first Wolfe condition, can always be satisfied by a sufficiently small step size along a descent direction.

```
function backtracking_line_search(f, ∇f, x, d, α; p=0.5, β=1e-4)
    y, g = f(x), ∇f(x)
    while f(x + α*d) > y + β*α*(g⋅d)
        α *= p
    end
    return α
end
```

Algorithm 4.3. The backtracking line search algorithm, which takes objective function `f`, its gradient `∇f`, the current design point `x`, a descent direction `d`, and the maximum step size `α`. We can optionally specify the reduction factor `p` and the first Wolfe condition parameter `β`. Note that the cdot character `⋅` aliases to the `dot` function such that `a⋅b` is equivalent to `dot(a,b)`. The symbol can be created by typing `\cdot` and hitting tab.

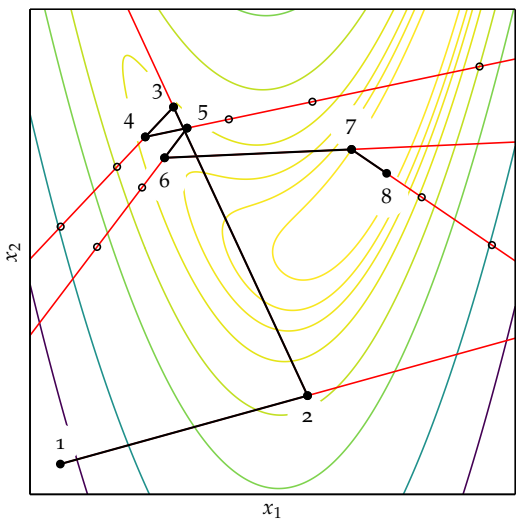


Figure 4.2. Backtracking line search used on the Rosenbrock function (appendix B.6). The black lines show the seven iterations taken by the descent method and the red lines show the points considered during each line search.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 4.2. An example of backtracking line search, an approximate line search method.

Consider approximate line search on $f(x_1, x_2) = x_1^2 + x_1 x_2 + x_2^2$ from $\mathbf{x} = [1, 2]$ in the direction $\mathbf{d} = [-1, -1]$, using a maximum step size of 10, a reduction factor of 0.5, and a first Wolfe condition parameter of $\beta = 10^{-4}$.

We check whether the maximum step size satisfies the first Wolfe condition, where the gradient at $\mathbf{x}$ is $\mathbf{g} = [4, 5]$:

$$\begin{aligned} f(\mathbf{x} + \alpha\mathbf{d}) &\leq f(\mathbf{x}) + \beta\alpha(\mathbf{g}^\top \mathbf{d}) \\ f([1, 2] + 10 \cdot [-1, -1]) &\leq 7 + 10^{-4} \cdot 10 \cdot [4, 5]^\top [-1, -1] \\ 217 &\leq 6.991 \end{aligned}$$

Because it is not satisfied, the step size is multiplied by 0.5 to obtain 5, and the first Wolfe condition is checked again:

$$\begin{aligned} f([1, 2] + 5 \cdot [-1, -1]) &\leq 7 + 10^{-4} \cdot 5 \cdot [4, 5]^\top [-1, -1] \\ 37 &\leq 6.996 \end{aligned}$$

Because it is not satisfied, the step size is multiplied by 0.5 to obtain 2.5, and the first Wolfe condition is checked again:

$$\begin{aligned} f([1, 2] + 2.5 \cdot [-1, -1]) &\leq 7 + 10^{-4} \cdot 2.5 \cdot [4, 5]^\top [-1, -1] \\ 3.25 &\leq 6.998 \end{aligned}$$

The first Wolfe condition is satisfied, and we terminate the line search.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

4.4.2 Curvature Condition

The second Wolfe condition, called the *curvature condition*, requires the directional derivative at the next iterate to be shallower (less negative):

$$\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k+1)}) \geq \sigma \nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) \tag{4.9}$$

where the parameter $\sigma \in (\beta, 1)$ controls how shallow the next directional derivative must be. The intuition is that we should probably take a larger step if we are still descending steeply so that we can get closer to a point where the first-order necessary condition for optimality is satisfied. [10]

[10] A typical value for σ is 0.1 when used with the conjugate gradient method and 0.9 when used with Newton's method. The conjugate gradient method is introduced in section 5.2, and Newton's method is introduced in section 6.1.

An alternative to the curvature condition is the *strong curvature condition*, which is a more restrictive criterion in that the slope is also required not to be too positive:

$$|\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k+1)})| \leq -\sigma \nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) \tag{4.10}$$

Figure 4.3 illustrates this condition.

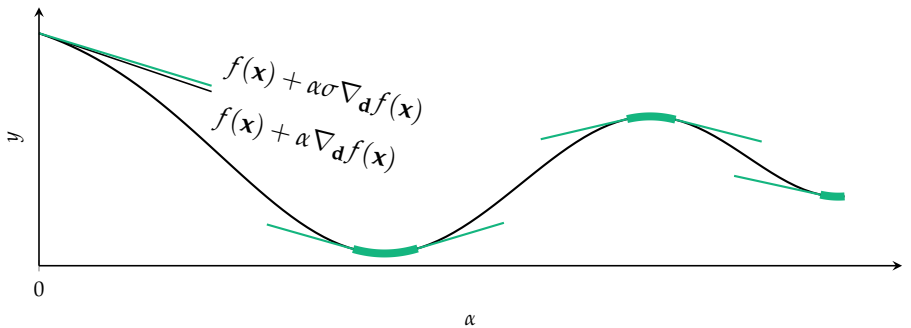


Figure 4.3. Regions where the strong curvature condition is satisfied.

Together, the sufficient decrease condition and the curvature condition form the Wolfe conditions. The sufficient decrease condition with the strong curvature condition form the *strong Wolfe conditions*. We can prove that step sizes that satisfy either set of Wolfe conditions always exist. Aside from edge cases such as happening to land at a saddle point, repeated line searches that always satisfy either set of Wolfe conditions are guaranteed to converge to a local minimum.[11]

[11] These proofs stipulate that the gradient always exists, is Lipschitz continuous, and that f is bounded below. For the original proofs, see P. Wolfe, "Convergence Conditions for Ascent Methods," *SIAM Review*, vol. 11, no. 2, pp. 226–235, 1969.

Satisfying the strong Wolfe conditions requires a more complicated algorithm called *strong backtracking line search* (algorithm 4.4).[12] The method operates in two phases. The first phase, the *bracketing phase*, tests successively larger step sizes to bracket an interval $[\alpha^{(k-1)}, \alpha^{(k)}]$ guaranteed to contain step lengths satisfying the strong Wolfe conditions.

[12] J. Nocedal and S.J. Wright, *Numerical Optimization*, 2nd ed. Springer, 2006.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function strong_backtracking(f, ∇f, x, d; α=1, β=1e-4, σ=0.1)
    y0, g0, y_prev, α_prev = f(x), ∇f(x)⋅d, NaN, 0
    αlo, αhi = NaN, NaN

    # bracket phase
    while true
        y = f(x + α*d)
        if y > y0 + β*α*g0 || (!isnan(y_prev) && y ≥ y_prev)
            αlo, αhi = α_prev, α
            break
        end
        g = ∇f(x + α*d)⋅d
        if abs(g) ≤ -σ*g0
            return α
        elseif g ≥ 0
            αlo, αhi = α, α_prev
            break
        end
        y_prev, α_prev, α = y, α, 2α
    end

    # zoom phase
    ylo = f(x + αlo*d)
    while true
        α = (αlo + αhi)/2
        y = f(x + α*d)
        if y > y0 + β*α*g0 || y ≥ ylo
            αhi = α
        else
            g = ∇f(x + α*d)⋅d
            if abs(g) ≤ -σ*g0
                return α
            elseif g*(αhi - αlo) ≥ 0
                αhi = αlo
            end
            αlo = α
        end
    end
end
```

Algorithm 4.4. Strong backtracking approximate line search for satisfying the strong Wolfe conditions. It takes as input the objective function `f`, the gradient function `∇f`, the design point `x` and direction `d` from which line search is conducted, an initial step size `α`, and the Wolfe condition parameters `β` and `σ`. The algorithm's bracket phase first brackets an interval containing a step size that satisfies the strong Wolfe conditions. It then reduces this bracketed interval in the zoom phase until a suitable step size is found. We interpolate with bisection, but other schemes can be used.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

An interval guaranteed to contain step lengths satisfying the strong Wolfe conditions is found when one of the following conditions hold:

$$f(\mathbf{x} + \alpha\mathbf{d}) \geq f(\mathbf{x}) \tag{4.11}$$

$$f(\mathbf{x} + \alpha\mathbf{d}) > f(\mathbf{x}) + \beta\alpha\nabla_{\mathbf{d}} f(\mathbf{x}) \tag{4.12}$$

$$\nabla_{\mathbf{d}} f(\mathbf{x} + \alpha\mathbf{d}) \geq 0 \tag{4.13}$$

Satisfying equation (4.12) is equivalent to violating the first Wolfe condition, thereby ensuring that shrinking the step length will guarantee a satisfactory step length. Similarly, equation (4.11) and equation (4.13) guarantee that the descent step has overshot a local minimum, and the region between must therefore contain a satisfactory step length. Figure 4.4 shows where each bracketing condition holds for an example line search.

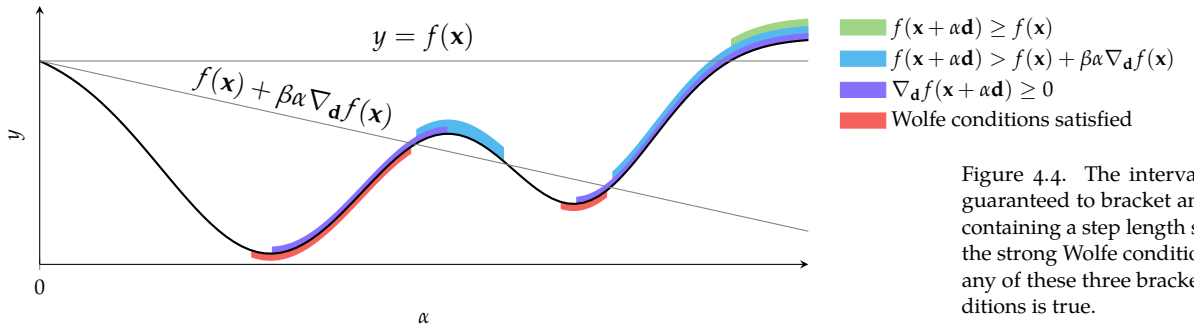


Figure 4.4. The interval $[0, \alpha]$ is guaranteed to bracket an interval containing a step length satisfying the strong Wolfe conditions when any of these three bracketing conditions is true.

The *zoom phase* shrinks the interval to find a step size satisfying the strong Wolfe conditions. The shrinking of the interval $[\alpha^{(k-1)}, \alpha^{(k)}]$ can be done using bisection. Figure 4.5 illustrates this process.

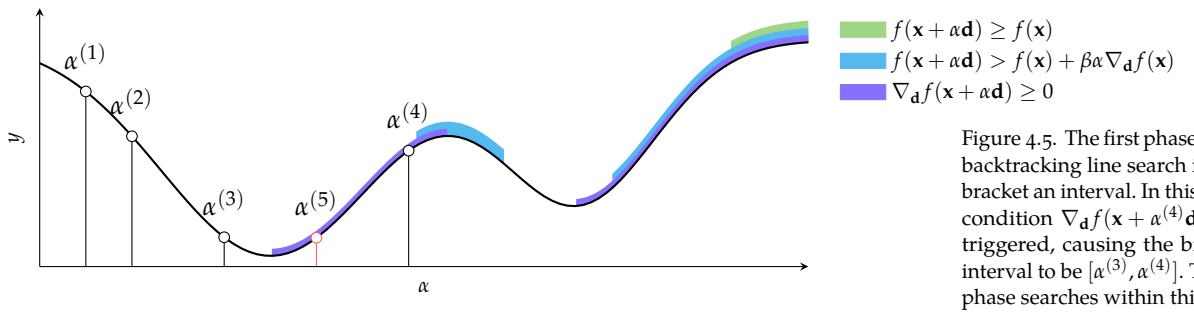


Figure 4.5. The first phase of strong backtracking line search is used to bracket an interval. In this case, the condition $\nabla_{\mathbf{d}} f(\mathbf{x} + \alpha^{(4)}\mathbf{d}) \geq 0$ is triggered, causing the bracketing interval to be $[\alpha^{(3)}, \alpha^{(4)}]$. The zoom phase searches within this interval using bisection to find a suitable step factor, resulting in $\alpha^{(5)}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

4.5 Trust Region Methods

Descent methods can place too much trust in their first- or second-order information, which can result in excessively large steps or premature convergence. A *trust region*[13] is the local area of the design space where the local model is believed to be reliable. A trust region method, or *restricted step method*, maintains a local model of the trust region that both limits the step taken by traditional line search and predicts the improvement associated with taking the step. If the improvement closely matches the predicted value, the trust region is expanded. If the improvement deviates from the predicted value, the trust region is contracted.[14] Figure 4.6 shows a design point centered within a circular trust region.

Trust region methods first choose the maximum step size and then the step direction, which is in contrast with line search methods that first choose a step direction, and then optimize the step size. A trust region approach finds the next step by minimizing a model of the objective function $\hat{f}$ over a trust region centered on the current design point $\mathbf{x}$. An example of $\hat{f}$ is a second-order Taylor approximation (see appendix C.2). The radius of the trust region, δ, is expanded and contracted based on how well the model predicts function evaluations. The next design point $\mathbf{x}'$ is obtained by solving:

$$\begin{aligned} &\underset{\mathbf{x}'}{\text{minimize}} && \hat{f}(\mathbf{x}') \\ &\text{subject to} && \left\|\mathbf{x} - \mathbf{x}'\right\| \leq \delta \end{aligned} \tag{4.14}$$

where the trust region is defined by the positive radius δ and a vector norm.[15] The equation above is a constrained optimization problem, which is covered in chapter 10.

The trust region radius δ is expanded or contracted based on the local model's predictive performance. Trust region methods compare the predicted improvement $\Delta y_{\text{pred}} = f(\mathbf{x}) - \hat{f}(\mathbf{x}')$ to the actual improvement $\Delta y_{\text{act}} = f(\mathbf{x}) - f(\mathbf{x}')$:

$$\eta = \frac{\text{actual improvement}}{\text{predicted improvement}} = \frac{f(\mathbf{x}) - f(\mathbf{x}')}{f(\mathbf{x}) - \hat{f}(\mathbf{x}')} \tag{4.15}$$

The ratio η is close to 1 when the predicted improvement matches the actual improvement. If the ratio is too small, such as below a threshold η_1, then the improvement is considered sufficiently less than expected, and the trust region radius is scaled down by a factor $\gamma_1 < 1$. If the ratio is sufficiently large, such

[13] K. Levenberg, "A Method for the Solution of Certain Non-Linear Problems in Least Squares," *Quarterly of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[14] A recent review of trust region methods is provided by Y. X. Yuan, "Recent Advances in Trust Region Algorithms," *Mathematical Programming*, vol. 151, no. 1, pp. 249–281, 2015.

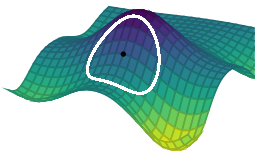


Figure 4.6. Trust region methods constrain the next step to lie within a local region. The trusted region is expanded and contracted based on the predictive performance of models of the objective function.

[15] There are a variety of methods for solving equation (4.14) efficiently. For an overview of the trust region method applied to quadratic models, see D. C. Sorensen, "Newton's Method with a Model Trust Region Modification," *SIAM Journal on Numerical Analysis*, vol. 19, no. 2, pp. 409–426, 1982.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

as above a threshold η_2, then our prediction is considered accurate, and the trust region radius is scaled up by a factor $\gamma_2 > 1$. Algorithm 4.5 provides an implementation and figure 4.7 demonstrates the optimization procedure.

```
mutable struct TrustRegionDescent <: DescentMethod
    δ  # trust region radius
    η1 # improvement ratio down-scale threshold
    η2 # improvement ratio up-scale threshold
    γ1 # down-scale multiplier
    γ2 # up-scale multiplier
end
function step!(M::TrustRegionDescent, f, ∇f, H, x)
    δ, η1, η2, γ1, γ2 = M.δ, M.η1, M.η2, M.γ1, M.γ2

    x′, y′ = solve_trust_region_subproblem(f, ∇f, H, x, δ)
    η = (f(x) - f(x′)) / (f(x) - y′) # improvement ratio
    if η < η1
        M.δ *= γ1 # scale down trust region
    else
        if η > η2
            M.δ *= γ2 # scale up trust region
        end
        return x′ # accept new point
    end
    return x
end

using Convex, ECOS
function solve_trust_region_subproblem(f, ∇f, H, x0, δ)
    x = Variable(length(x0))
    f = f(x0) + ∇f(x0)⋅(x-x0) + quadform(x-x0, H(x0))/2
    p = minimize(f, norm(x-x0) ≤ δ)
    solve!(p, ECOS.Optimizer)
    return (x.value, p.optval)
end
```

Algorithm 4.5. The trust region descent method, where `f` is the objective function, `∇f` produces the derivative, `H` produces the Hessian, and `x` is the current design point. The parameters `η1` and `η2` determine when the trust region radius `δ` is increased or decreased, and `γ1` and `γ2` control the magnitude of the change. An implementation for `solve_trust_region_subproblem` must be provided that solves equation (4.14). We have provided an example implementation that uses a second-order Taylor approximation about `x0` with a circular trust region that assumes the Hessian is positive definite.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

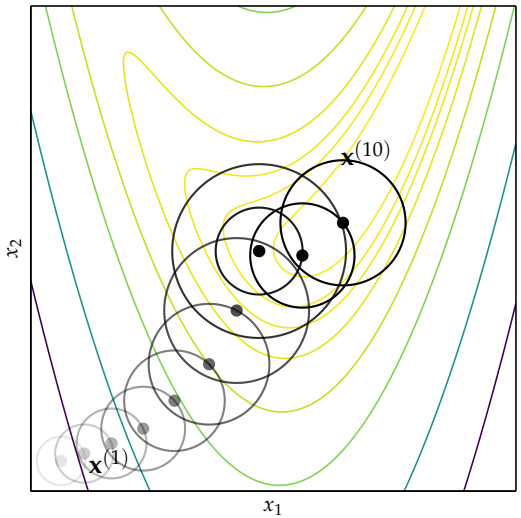


Figure 4.7. Trust region optimization used on the Rosenbrock function (appendix B.6).

Trust regions need not be circular. In some cases, certain directions may have higher trust than others. A norm can be constructed to produce elliptical regions as shown in figure 4.8:

$$\|\mathbf{x} - \mathbf{x}_0\|_{\mathbf{E}} = \sqrt{(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{E} (\mathbf{x} - \mathbf{x}_0)} \tag{4.16}$$

The ellipse matrix $\mathbf{E}$ can be updated with each descent iteration, which can involve more complicated adjustments than scaling the trusted region.[16]

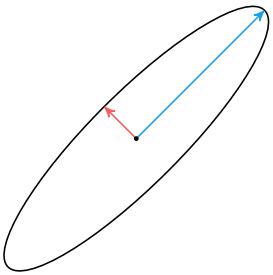


Figure 4.8. Trust region optimization can be adaptive to use elliptical trust regions. In this case, the trust region is elongated in the direction of the blue arrow and contracted in the direction of the red arrow.

4.6 Termination Conditions

There are four common termination conditions for descent direction methods:

- *Maximum iterations.* We may want to terminate when the number of iterations k exceeds some threshold $k_{\max}$. Alternatively, we might want to terminate once a maximum amount of elapsed time is exceeded.

$$k > k_{\max} \tag{4.17}$$

[16] Additional detail is provided by J. Nocedal and S. J. Wright, "Trust-Region Methods," in *Numerical Optimization*, T. V. Mikosch, S. I. Resnick, and S. M. Robinson, eds. Springer, 2006, pp. 66–100.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- *Absolute improvement.* This termination condition looks at the change in the function value over subsequent steps. If the change is smaller than a given threshold, it will terminate:

$$f(\mathbf{x}^{(k)}) - f(\mathbf{x}^{(k+1)}) < \epsilon_a \tag{4.18}$$

- *Relative improvement.* This termination condition also looks at the change in function value but uses the change relative to the current function value:

$$f(\mathbf{x}^{(k)}) - f(\mathbf{x}^{(k+1)}) < \epsilon_r |f(\mathbf{x}^{(k)})| \tag{4.19}$$

- *Gradient magnitude.* We can also terminate based on the magnitude of the gradient:

$$\|\nabla f(\mathbf{x}^{(k+1)})\| < \epsilon_g \tag{4.20}$$

In cases where multiple local minima are likely to exist, it can be beneficial to incorporate *random restarts* after our termination conditions are met where we restart our local descent method from randomly selected initial points.

4.7 Summary

- Descent direction methods incrementally descend toward a local optimum.
- Univariate optimization can be applied during line search.
- Approximate line search can be used to identify appropriate descent step sizes.
- Trust region methods constrain the step to lie within a local region that expands or contracts based on predictive accuracy.
- Termination conditions for descent methods can be based on criteria such as the change in the objective function value or magnitude of the gradient.

4.8 Exercises

Exercise 4.1. Why is it important to have more than one termination condition?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: Consider running a descent method on $f(x) = 1/x$ for $x > 0$. The minimum does not exist, and the descent method will forever proceed in the positive x direction with ever-increasing step sizes. Thus, only relying on a step-size termination condition would cause the method to run forever. Also terminating based on gradient magnitude would cause it to terminate.

A descent method applied to $f(x) = -x$ will also forever proceed in the positive x direction. The function is unbounded below, so neither a step-size termination condition nor a gradient magnitude termination condition would trigger. It is common to include an additional termination condition to limit the number of iterations.

Exercise 4.2. The first Wolfe condition requires

$$f(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}) \leq f(\mathbf{x}^{(k)}) + \beta\alpha\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)})$$

What is the maximum step length α that satisfies this condition, given that $f(\mathbf{x}) = 5 + x_1^2 + x_2^2$, $\mathbf{x}^{(k)} = [-1, -1]$, $\mathbf{d} = [1, 0]$, and $\beta = 10^{-4}$?

Solution: Applying the first Wolfe condition to our objective function yields $6 + (-1 + \alpha)^2 \leq 7 - 2\alpha \cdot 10^{-4}$, which can be simplified to $\alpha^2 - 2\alpha + 2 \cdot 10^{-4}\alpha \leq 0$. This equation can be solved to obtain $\alpha \leq 2(1 - 10^{-4})$. Thus, the maximum step length is $\alpha = 1.9998$.

Exercise 4.3. The first Wolfe condition is parameterized by a $\beta \in [0, 1)$. Show that using $\beta = 1$ can cause there to be no valid positive steps by applying the first Wolfe condition to minimizing $f(x) = x^2$ for some $d^{(k)} \neq 0$ and $x^{(k)}$.

Solution: We substitute our objective function into the first Wolfe condition:

$$\begin{aligned}
f(x^{(k)} + \alpha d^{(k)}) &\leq f(x^{(k)}) + \beta\alpha\nabla_{d^{(k)}} f(x^{(k)}) \\
\left(x^{(k)} + \alpha d^{(k)}\right)^2 &\leq \left(x^{(k)}\right)^2 + 2\beta\alpha d^{(k)} x^{(k)} \\
\left(x^{(k)}\right)^2 + 2\alpha d^{(k)} x^{(k)} + \alpha^2\left(d^{(k)}\right)^2 &\leq \left(x^{(k)}\right)^2 + 2\beta\alpha d^{(k)} x^{(k)} \\
2\alpha d^{(k)} x^{(k)} + \alpha^2\left(d^{(k)}\right)^2 &\leq 2\beta\alpha d^{(k)} x^{(k)}
\end{aligned}$$

Substituting $\beta = 1$ results in $\alpha^2\left(d^{(k)}\right)^2 \leq 0$, and thus the only valid step factor is $\alpha = 0$, which is not a positive step.

Exercise 4.4. Second-order methods, covered in chapter 6, use curvature information to perform updates or calculate descent directions. These methods often benefit from positive curvature. How would the Wolfe conditions help line search produce iterates with positive curvature?

Solution: The first Wolfe condition does not directly influence curvature. The second Wolfe condition ensures that the chosen step results in a shallower successor. This directly relates to positive curvature, as positive curvature is the increase of the directional derivative in a given direction.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

5 First-Order Methods

The previous chapter introduced the general concept of descent direction methods. This chapter discusses a variety of algorithms that use *first-order* methods to select the appropriate descent direction. First-order methods rely on gradient information to help direct the search for a minimum, which can be obtained using methods outlined in chapter 2.

5.1 Gradient Descent

The *gradient descent* method[1] uses the gradient to select the next descent direction $\mathbf{d}$. For convenience, we define the gradient at the kth iterate $\mathbf{x}^{(k)}$ to be

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)}) \tag{5.1}$$

[1] The gradient is only available when f is differentiable. It is sometimes possible to define alternatives for non-differentiable functions (appendix C.5).

The motivation for gradient descent comes from the first-order Taylor series approximation about our current iterate $\mathbf{x}^{(k)}$:

$$f(\mathbf{x}^{(k)} + \alpha\mathbf{d}) \approx f(\mathbf{x}^{(k)}) + \alpha\mathbf{d}^\top\mathbf{g}^{(k)} \tag{5.2}$$

We can choose the direction $\mathbf{d}$ that minimizes this first-order approximation subject to the constraint that $\|\mathbf{d}\| = 1$.[2] The direction that minimizes this first-order approximation is the direction of *steepest descent*, which is simply the direction opposite the gradient:

$$\mathbf{d}^{(k)} = -\frac{\mathbf{g}^{(k)}}{\|\mathbf{g}^{(k)}\|} \tag{5.3}$$

Some implementations of gradient descent do not normalize the descent direction. In that case, the step factor α does not correspond to the step length.

[2] We assume the L_2 norm here, but steepest descent with respect to other norms corresponds to other optimization methods. For example, the L_1 norm leads to the *coordinate descent* method. See Section 9.4 of S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

Following the direction of steepest descent is guaranteed to lead to improvement, provided that the objective function is smooth, the step factor α is sufficiently small, and we are not already at a point where the gradient is zero.[3] Our intuition might suggest that we cannot do better than going in the direction of steepest descent, but this is not the case, even when we optimize the step size. Choosing step sizes that maximally decrease f produces jagged search paths. The next descent direction will always be orthogonal to the current direction, as shown in figure 5.1.

[3] A point where the gradient is zero is called a *stationary point*.

We can show that the next direction is orthogonal to the current direction as follows. If we optimize the step size at each step, we have

$$\alpha^{(k)} = \arg\min_{\alpha} f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)}) \tag{5.4}$$

The optimization above implies that the directional derivative equals zero. Using equation (2.9), we have

$$\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})^\top \mathbf{d}^{(k)} = 0 \tag{5.5}$$

We know

$$\mathbf{d}^{(k+1)} = -\frac{\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})}{\|\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})\|} \tag{5.6}$$

Hence,

$$\mathbf{d}^{(k+1)\top} \mathbf{d}^{(k)} = 0 \tag{5.7}$$

which means that $\mathbf{d}^{(k+1)}$ and $\mathbf{d}^{(k)}$ are orthogonal.

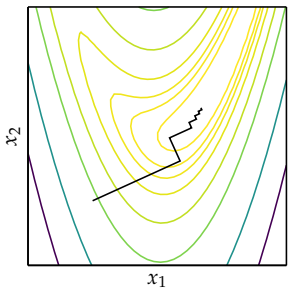


Figure 5.1. Gradient descent can result in zig-zagging in narrow canyons as shown on the Rosenbrock function (appendix B.6).

Narrow valleys aligned with a descent direction are not an issue. When narrow valleys are not aligned with the descent direction,[4] many steps must be taken in order to make progress along the valley's floor as shown in figure 5.1. An implementation of gradient descent is provided by algorithm 5.1.

[4] Optimization methods can be sensitive to the scaling of the design variables, especially methods that depend upon the gradient. See, for example, exercise 5.11.

```
struct GradientDescent <: DescentMethod
    α # step factor
end
init!(M::GradientDescent, f, ∇f, x) = M
function step!(M::GradientDescent, f, ∇f, x)
    α, g = M.α, ∇f(x)
    return x - α*g
end
```

Algorithm 5.1. The gradient descent method, which follows the direction of gradient descent with a fixed step factor. The `step!` function produces the next iterate whereas the `init!` function does nothing.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

5.2 *Conjugate Gradient*

Gradient descent can perform poorly in narrow valleys. The *conjugate gradient* method overcomes this issue by borrowing inspiration from methods for optimizing quadratic functions:

$$\underset{\mathbf{x}}{\text{minimize}}\ \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c \tag{5.8}$$

where $\mathbf{A}$ is symmetric and positive definite, and thus the problem has a unique local minimum (section 1.6.2).

The conjugate gradient method can optimize n-dimensional quadratic functions in n steps as shown in figure 5.2. Its directions $\mathbf{d}^{(1)}, \ldots, \mathbf{d}^{(n)} \neq \mathbf{0}$ are *mutually conjugate* with respect to $\mathbf{A}$:

$$\mathbf{d}^{(i)\top} \mathbf{A}\, \mathbf{d}^{(j)} = 0 \ \text{ for all } \ i \neq j \tag{5.9}$$

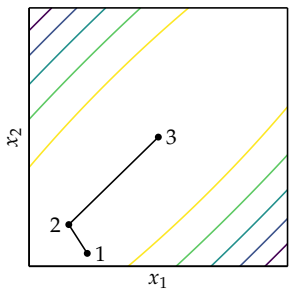


Figure 5.2. Conjugate gradient descent converges in n steps when applied to an n-dimensional quadratic function.

The n mutually conjugate vectors form a basis of $\mathbb{R}^n$. They are generally not orthogonal to one another.

The successive conjugate directions are computed using gradient information and the previous descent direction. The algorithm starts with the direction of steepest descent:

$$\mathbf{d}^{(1)} = -\mathbf{g}^{(1)} \tag{5.10}$$

We then use line search to find the next design point. For quadratic functions, the step factor α can be computed exactly (example 5.1). The update is then:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \alpha^{(1)}\mathbf{d}^{(1)} \tag{5.11}$$

Subsequent iterations choose $\mathbf{d}$ based on the current gradient and a contribution from the previous descent direction:

$$\mathbf{d}^{(k)} = -\mathbf{g}^{(k)} + \beta^{(k)}\mathbf{d}^{(k-1)} \tag{5.12}$$

for scalar parameter $\beta^{(k)}$. Larger values of $\beta^{(k)}$ indicate that the previous descent direction contributes more strongly.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 5.1. The optimal step factor for a line search on a quadratic function.

Suppose we want to derive the optimal step factor for a line search on a quadratic function:

$$\underset{\alpha}{\text{minimize}}\, f(\mathbf{x} + \alpha\mathbf{d})$$

We can compute the derivative with respect to α:

$$\begin{aligned}\frac{\partial f(\mathbf{x}+\alpha\mathbf{d})}{\partial\alpha} &= \frac{\partial}{\partial\alpha}\left[\frac{1}{2}(\mathbf{x}+\alpha\mathbf{d})^\top\mathbf{A}(\mathbf{x}+\alpha\mathbf{d}) + \mathbf{b}^\top(\mathbf{x}+\alpha\mathbf{d}) + c\right] \\ &= \mathbf{d}^\top\mathbf{A}(\mathbf{x}+\alpha\mathbf{d}) + \mathbf{d}^\top\mathbf{b} \\ &= \mathbf{d}^\top(\mathbf{A}\mathbf{x}+\mathbf{b}) + \alpha\mathbf{d}^\top\mathbf{A}\mathbf{d}\end{aligned}$$

Setting $\frac{\partial f(\mathbf{x}+\alpha\mathbf{d})}{\partial\alpha} = 0$ results in:

$$\alpha = -\frac{\mathbf{d}^\top(\mathbf{A}\mathbf{x}+\mathbf{b})}{\mathbf{d}^\top\mathbf{A}\mathbf{d}}$$

We can derive the best value for $\beta^{(k)}$ for a known $\mathbf{A}$, using the fact that $\mathbf{d}^{(k)}$ is conjugate to $\mathbf{d}^{(k-1)}$:

$$\mathbf{d}^{(k)\top}\mathbf{A}\mathbf{d}^{(k-1)} = 0 \tag{5.13}$$

$$\Rightarrow (-\mathbf{g}^{(k)} + \beta^{(k)}\mathbf{d}^{(k-1)})^\top\mathbf{A}\mathbf{d}^{(k-1)} = 0 \tag{5.14}$$

$$\Rightarrow -\mathbf{g}^{(k)\top}\mathbf{A}\mathbf{d}^{(k-1)} + \beta^{(k)}\mathbf{d}^{(k-1)\top}\mathbf{A}\mathbf{d}^{(k-1)} = 0 \tag{5.15}$$

$$\Rightarrow \beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\mathbf{A}\mathbf{d}^{(k-1)}}{\mathbf{d}^{(k-1)\top}\mathbf{A}\mathbf{d}^{(k-1)}} \tag{5.16}$$

The conjugate gradient method can be applied to nonquadratic functions as well. Smooth functions behave like quadratic functions close to a local minimum, and the conjugate gradient method will converge quickly in such regions. Unfortunately, we do not know the value of $\mathbf{A}$ that best approximates f around $\mathbf{x}^{(k)}$. Several updates have been proposed for $\beta^{(k)}$ that do not require knowing $\mathbf{A}$ and tend to work well. One is the *Fletcher-Reeves*[5] update:

$$\beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\mathbf{g}^{(k)}}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}} \tag{5.17}$$

This update is equivalent to setting $\beta^{(k)}$ to the ratio of the squared norm of the current gradient to the squared norm of the previous gradient.

[5] R. Fletcher and C. M. Reeves, "Function Minimization by Conjugate Gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 1964.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

An alternative to the Fletcher-Reeves update is the *Polak-Ribière*[6] update:

$$\beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\left(\mathbf{g}^{(k)} - \mathbf{g}^{(k-1)}\right)}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}} \tag{5.18}$$

Convergence for the Polak-Ribière method (algorithm 5.2) can be guaranteed if we modify it to allow for automatic resets:

$$\beta \leftarrow \max(\beta, 0) \tag{5.19}$$

[6] E. Polak and G. Ribière, "Note sur la Convergence de Méthodes de Directions Conjuguées," *Revue Française d'informatique et de Recherche Opérationnelle, Série Rouge*, vol. 3, no. 1, pp. 35–43, 1969.

```
mutable struct ConjugateGradientDescent <: DescentMethod
    d # previous search direction
    g # previous gradient
end
function init!(M::ConjugateGradientDescent, f, ∇f, x)
    M.g = ∇f(x)
    M.d = -M.g
    return M
end
function step!(M::ConjugateGradientDescent, f, ∇f, x)
    d, g = M.d, M.g
    g′ = ∇f(x)
    β = max(0, g′⋅(g′-g)/(g⋅g))
    d′ = -g′ + β*d
    x′ = line_search(f, x, d′)
    M.d, M.g = d′, g′
    return x′
end
```

Algorithm 5.2. The conjugate gradient method with the Polak-Ribière update, where `d` is the previous search direction and `g` is the previous gradient.

Figure 5.3 shows an example search using this method.

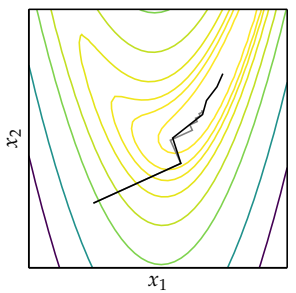


Figure 5.3. The conjugate gradient method with the Polak-Ribière update. Gradient descent is shown in gray.

5.3 *Momentum*

Gradient descent will take a long time to traverse a nearly flat surface as shown in figure 5.4. Allowing momentum to accumulate is one way to speed progress. We can modify gradient descent to incorporate momentum. In addition to tracking the design iterate $\mathbf{x}^{(k)}$, we also track its associated velocity vector $\mathbf{v}^{(k)}$.

The *momentum* update equations are:

$$\mathbf{v}^{(k+1)} = \beta\mathbf{v}^{(k)} - \alpha\mathbf{g}^{(k)} \tag{5.20}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)} \tag{5.21}$$

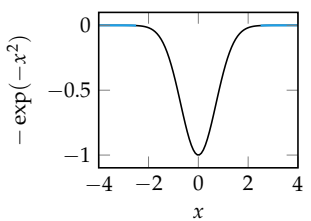


Figure 5.4. Regions that are nearly flat have gradients with small magnitudes and can thus require many iterations of gradient descent to traverse.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

For $\beta = 0$, we recover gradient descent. Momentum can be interpreted as a ball rolling down a nearly horizontal incline. The ball naturally gathers momentum as gravity causes it to accelerate, just as the gradient causes momentum to accumulate in this descent method. Momentum descent is compared to gradient descent in figure 5.5. An implementation is provided in algorithm 5.3.

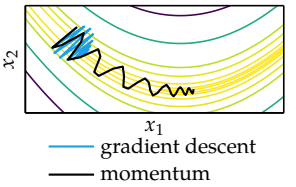


Figure 5.5. Gradient descent and the momentum method compared on the Rosenbrock function with $b = 100$; see appendix B.6.

```
mutable struct Momentum <: DescentMethod
    α # step factor
    β # momentum decay
    v # momentum
end
function init!(M::Momentum, f, ∇f, x)
    M.v = zero(x)
    return M
end
function step!(M::Momentum, f, ∇f, x)
    α, β, v, g = M.α, M.β, M.v, ∇f(x)
    v .= β*v - α*g
    return x + v
end
```

Algorithm 5.3. The momentum method for accelerated descent. The first line in `step!` makes copies of the scalars `α` and `β`, but creates a reference to the vector `v`. The line `v .= β*v - α*g` modifies the original momentum vector in the struct `M`. See appendix A for a review of array assignment, referencing, and copying.

5.4 Nesterov Momentum

One issue of momentum is that the steps do not slow down enough at the bottom of a valley and tend to overshoot the valley floor. *Nesterov momentum*[7] modifies the momentum algorithm to use the gradient at the projected future position:

$$\mathbf{v}^{(k+1)} = \beta\mathbf{v}^{(k)} - \alpha\nabla f(\mathbf{x}^{(k)} + \beta\mathbf{v}^{(k)}) \tag{5.22}$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)} \tag{5.23}$$

The Nesterov momentum and momentum descent methods are compared in figure 5.6. An implementation is provided by algorithm 5.4.

[7] Y. Nesterov, "A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$," *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 543–547, 1983.

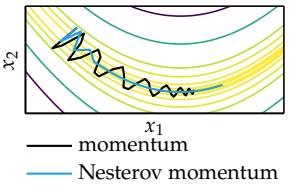


Figure 5.6. The momentum and Nesterov momentum methods compared on the Rosenbrock function with $b = 100$.

5.5 AdaGrad

Momentum and Nesterov momentum update all components of $\mathbf{x}$ with the same step factor. The *adaptive gradient* method, or *AdaGrad*,[8] adapts a step factor for each component of $\mathbf{x}$. AdaGrad updates parameters with smaller or infrequently

[8] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct NesterovMomentum <: DescentMethod
    α # step factor
    β # momentum decay
    v # momentum
end
function init!(M::NesterovMomentum, f, ∇f, x)
    M.v = zero(x)
    return M
end
function step!(M::NesterovMomentum, f, ∇f, x)
    α, β, v = M.α, M.β, M.v
    v .= β*v - α*∇f(x + β*v)
    return x + v
end
```

Algorithm 5.4. Nesterov's momentum method of accelerated descent.

large gradients more aggressively than parameters that are frequently subject to large gradients.[9]

AdaGrad updates have the form:

$$x_i^{(k+1)} = x_i^{(k)} - \alpha_{\text{adagrad},i}^{(k)} g_i^{(k)} \tag{5.24}$$

with componentwise step factors:

$$\alpha_{\text{adagrad},i}^{(k)} = \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} \tag{5.25}$$

In this equation, $s_i^{(k)}$ is the sum of the squares of the partial derivatives, with respect to x_i, up to time step k,

$$s_i^{(k)} = \sum_{j=1}^{k} \left(g_i^{(j)}\right)^2 \tag{5.26}$$

and ϵ is a small value, on the order of 10^{-8}, to prevent division by zero. Components with larger partial derivatives will thus receive smaller step factors.

AdaGrad is far less sensitive to the baseline step factor α, which is often set to a default value of 0.01. AdaGrad's primary weakness is that the components of $\mathbf{s}$ are each strictly nondecreasing. The accumulated sum causes the effective step factor to decrease during training, often becoming infinitesimally small before convergence. An implementation is provided by algorithm 5.5.

[9] AdaGrad is designed to work particularly well when the gradient is sparse. Many deep learning problems result in sparse gradients of the objective function as a result of some features occurring far less frequently than others.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct AdaGrad <: DescentMethod
    α # baseline step factor
    ϵ # small value
    s # sum of squared gradient
end
function init!(M::AdaGrad, f, ∇f, x)
    M.s = zero(x)
    return M
end
function step!(M::AdaGrad, f, ∇f, x)
    α, ϵ, s, g = M.α, M.ϵ, M.s, ∇f(x)
    s .+= g.*g
    return x - α*g ./ (sqrt.(s) .+ ϵ)
end
```

Algorithm 5.5. The AdaGrad accelerated descent method.

5.6 RMSProp

RMSProp (algorithm 5.6)[10] extends AdaGrad to avoid the effect of a monotonically decreasing step factor. RMSProp maintains a decaying average of squared gradients. This average is updated according to:[11]

$$\hat{\mathbf{s}}^{(k+1)} = \gamma\hat{\mathbf{s}}^{(k)} + (1-\gamma)\left(\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\right) \tag{5.27}$$

where the decay $\gamma \in [0, 1]$ is typically close to 0.9. The decaying average of past squared gradients can be substituted into RMSProp's update equation:[12]

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{\hat{s}_i^{(k+1)}}} g_i^{(k)} = x_i^{(k)} - \frac{\alpha}{\epsilon + \text{RMS}(g_i)} g_i^{(k)} \tag{5.28}$$

[10] RMSProp is unpublished and comes from Lecture 6e of Geoff Hinton's Coursera class.

[11] The operation $\mathbf{a} \odot \mathbf{b}$ is the element-wise product between vectors $\mathbf{a}$ and $\mathbf{b}$.

[12] The denominator is similar to the root mean square (RMS) of the gradient component. In this chapter we use $\text{RMS}(x)$ to refer to the decaying root mean square of the time series of x.

5.7 Adadelta

Adadelta (algorithm 5.7)[13] is another method for overcoming AdaGrad's monotonically decreasing step factor. After independently deriving the RMSProp update, the authors noticed that the units in the update equations for gradient descent, momentum, and AdaGrad do not match. To fix this, they use an exponentially decaying average of the square updates:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\text{RMS}(\Delta x_i)}{\epsilon + \text{RMS}(g_i)} g_i^{(k)} \tag{5.29}$$

which eliminates the step factor parameter entirely.

[13] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," 2012. arXiv: 1212.5701.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct RMSProp <: DescentMethod
    α # step factor
    γ # decay
    ϵ # small value
    s # sum of squared gradient
end
function init!(M::RMSProp, f, ∇f, x)
    M.s = zero(x)
    return M
end
function step!(M::RMSProp, f, ∇f, x)
    α, γ, ϵ, s, g = M.α, M.γ, M.ϵ, M.s, ∇f(x)
    s .= γ*s + (1-γ)*(g.*g)
    return x - α*g ./ (sqrt.(s) .+ ϵ)
end
```

Algorithm 5.6. The RMSProp accelerated descent method.

```
mutable struct Adadelta <: DescentMethod
    γs # gradient decay
    γx # update decay
    ϵ  # small value
    s  # sum of squared gradients
    u  # sum of squared updates
end
function init!(M::Adadelta, f, ∇f, x)
    M.s = zero(x)
    M.u = zero(x)
    return M
end
function step!(M::Adadelta, f, ∇f, x)
    γs, γx, ϵ, s, u, g = M.γs, M.γx, M.ϵ, M.s, M.u, ∇f(x)
    s .= γs*s + (1-γs)*g.*g
    Δx = - (sqrt.(u) .+ ϵ) ./ (sqrt.(s) .+ ϵ) .* g
    u .= γx*u + (1-γx)*Δx.*Δx
    return x + Δx
end
```

Algorithm 5.7. The Adadelta accelerated descent method. The small constant ϵ is added to the numerator as well to prevent progress from entirely decaying to zero and to start off the first iteration where $\Delta x = 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

5.8 Adam

The *adaptive moment estimation* method, or *Adam*,[14] also adapts step factors to each parameter (algorithm 5.8). It stores an exponentially decaying squared gradient like RMSProp and Adadelta, as well as an exponentially decaying gradient like momentum.

[14] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

Initializing the gradient and squared gradient to zero introduces a bias. A bias correction step helps alleviate the issue.[15] The equations applied during each iteration for Adam are:

[15] According to the original paper, good default settings are $\alpha = 0.001$, $\gamma_v = 0.9$, $\gamma_s = 0.999$, and $\epsilon = 10^{-8}$.

$$\text{biased decaying momentum: } \mathbf{v}^{(k+1)} = \gamma_v \mathbf{v}^{(k)} + (1 - \gamma_v)\mathbf{g}^{(k)} \tag{5.30}$$

$$\text{biased decaying sq. gradient: } \mathbf{s}^{(k+1)} = \gamma_s \mathbf{s}^{(k)} + (1 - \gamma_s)\left(\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\right) \tag{5.31}$$

$$\text{corrected decaying momentum: } \hat{\mathbf{v}}^{(k+1)} = \mathbf{v}^{(k+1)} / (1 - \gamma_v^k) \tag{5.32}$$

$$\text{corrected decaying sq. gradient: } \hat{\mathbf{s}}^{(k+1)} = \mathbf{s}^{(k+1)} / (1 - \gamma_s^k) \tag{5.33}$$

$$\text{next iterate: } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \hat{\mathbf{v}}^{(k+1)} / \left(\epsilon + \sqrt{\hat{\mathbf{s}}^{(k+1)}}\right) \tag{5.34}$$

5.9 Hypergradient Descent

The accelerated descent methods are either extremely sensitive to the step factor or go to great lengths to adapt the step factor during execution. The step factor dictates how sensitive the method is to the gradient signal. A rate that is too high or too low often drastically affects performance.

Hypergradient descent[16] was developed with the understanding that the derivative with respect to the step factor should be useful for improving optimizer performance. A *hypergradient* is a derivative taken with respect to a hyperparameter. Hypergradient algorithms reduce the sensitivity to the hyperparameter, allowing it to adapt more quickly.

[16] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online Learning Rate Adaptation with Hypergradient Descent," in *International Conference on Learning Representations (ICLR)*, 2018.

Hypergradient descent applies gradient descent to the step factor of an underlying descent method. The method requires the partial derivative of the objective function with respect to the step factor. For gradient descent, this partial derivative

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct Adam <: DescentMethod
    α  # step factor
    γv # decay
    γs # decay
    ϵ  # small value
    k  # step counter
    v  # 1st moment estimate
    s  # 2nd moment estimate
end
function init!(M::Adam, f, ∇f, x)
    M.k = 0
    M.v = zero(x)
    M.s = zero(x)
    return M
end
function step!(M::Adam, f, ∇f, x)
    α, γv, γs, ϵ, k = M.α, M.γv, M.γs, M.ϵ, M.k
    s, v, g = M.s, M.v, ∇f(x)
    v .= γv*v + (1-γv)*g
    s .= γs*s + (1-γs)*g.*g
    M.k = k += 1
    v_hat = v ./ (1 - γv^k)
    s_hat = s ./ (1 - γs^k)
    return x - α*v_hat ./ (sqrt.(s_hat) .+ ϵ)
end
```

Algorithm 5.8. The Adam accelerated descent method.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

is:

$$\frac{\partial f(\mathbf{x}^{(k+1)})}{\partial \alpha^{(k)}} = (\mathbf{g}^{(k+1)})^\top \frac{\partial}{\partial \alpha^{(k)}} \left(\mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{g}^{(k)}\right) \tag{5.35}$$

$$= (\mathbf{g}^{(k+1)})^\top \left(-\mathbf{g}^{(k)}\right) \tag{5.36}$$

Computing the hypergradient thus requires keeping track of the last gradient. The resulting update rule is:

$$\alpha^{(k)} = \alpha^{(k-1)} - \mu \frac{\partial f(\mathbf{x}^{(k)})}{\partial \alpha^{(k-1)}} \tag{5.37}$$

$$= \alpha^{(k-1)} + \mu (\mathbf{g}^{(k)})^\top \mathbf{g}^{(k-1)} \tag{5.38}$$

where μ is the hypergradient step factor.

This derivation can be applied to any gradient-based descent method that follows equation (4.1). These methods are visualized in figure 5.7. Implementations are provided for the hypergradient versions of gradient descent (algorithm 5.9) and Nesterov momentum (algorithm 5.10).

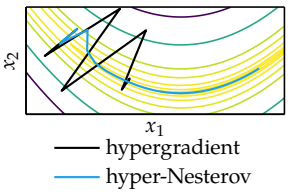


Figure 5.7. Hypergradient versions of gradient descent and Nesterov momentum compared on the Rosenbrock function with $b = 100$; see appendix B.6.

Algorithm 5.9. The hypergradient form of gradient descent.

```
mutable struct HyperGradientDescent <: DescentMethod
    α0     # initial step factor
    μ      # step factor for the step factor update
    α      # current step factor
    g_prev # previous gradient
end
function init!(M::HyperGradientDescent, f, ∇f, x)
    M.α = M.α0
    M.g_prev = zero(x)
    return M
end
function step!(M::HyperGradientDescent, f, ∇f, x)
    α, μ, g, g_prev = M.α, M.μ, ∇f(x), M.g_prev
    α = α + μ*(g⋅g_prev)
    M.g_prev, M.α = g, α
    return x - α*g
end
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct HyperNesterovMomentum <: DescentMethod
    α0     # initial step factor
    μ      # step factor for the step factor update
    β      # momentum decay
    v      # momentum
    α      # current step factor
    h_prev # previous gradient
end
function init!(M::HyperNesterovMomentum, f, ∇f, x)
    M.α = M.α0
    M.v = zero(x)
    M.h_prev = zero(x)
    return M
end
function step!(M::HyperNesterovMomentum, f, ∇f, x)
    α, β, μ, v = M.α, M.β, M.μ, M.v
    h, h_prev = ∇f(x + β*v), M.h_prev
    α += μ*(h⋅h_prev)
    v .= β*v - α*h
    M.h_prev, M.α = h, α
    return x + v
end
```

Algorithm 5.10. The hypergradient form of the Nesterov momentum descent method.

5.10 Summary

- Gradient descent follows the direction of steepest descent.
- The conjugate gradient method can automatically adjust to local valleys.
- Descent methods with momentum build up progress in favorable directions.
- A wide variety of accelerated descent methods use special techniques to speed up descent.
- Hypergradient descent applies gradient descent to the step factor of an underlying descent method.

5.11 Exercises

Exercise 5.1. Suppose we have $f(\mathbf{x}) = x_1 x_2^2$. For $\mathbf{x}^{(k)} = [1, 2]$, compute the normalized direction used for gradient descent.

Solution: The gradient is $\nabla f = [x_2^2, 2x_1x_2]$. At $\mathbf{x}^{(k)} = [1, 2]$, we get an unnormalized direction of steepest descent of $\mathbf{d} = [-4, -4]$, which is normalized to $\mathbf{d} = [-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 5.2. Compute the gradient of $\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x}$ when $\mathbf{A}$ is symmetric.

Solution: $\nabla f(\mathbf{x}) = 2\mathbf{A}\mathbf{x} + \mathbf{b}$.

Exercise 5.3. Apply gradient descent with a unit step factor to $f(x) = x^4$ from a starting point of your choice. Compute two iterations.

Solution: The derivative is $f'(x) = 4x^3$. Starting from $x^{(1)} = 1$:

$$\begin{aligned} f'(1) &= 4 && \rightarrow x^{(2)} = 1 - 4 = -3 \\ f'(-3) &= 4 \cdot (-27) = -108 && \rightarrow x^{(3)} = -3 + 108 = 105 \end{aligned}$$

Exercise 5.4. Apply one step of gradient descent to $f(x) = e^x + e^{-x}$ from $x^{(1)} = 10$ with both a unit step factor and with exact line search.

Solution: We have $f'(x) = e^x - e^{-x} \approx e^x$ for large x. Thus $f'(x^{(1)}) \approx e^{10}$ and $x^{(2)} \approx -e^{10}$. If we apply an exact line search, $x^{(2)} = 0$. Thus, without a line search we are not guaranteed to reduce the value of the objective function.

Exercise 5.5. The conjugate gradient method can also be used to find a search direction $\mathbf{d}$ when a local quadratic model of a function is available at the current point. With $\mathbf{d}$ as search direction, let the model be

$$q(\mathbf{d}) = \mathbf{d}^\top \mathbf{H}\mathbf{d} + \mathbf{b}^\top \mathbf{d} + \mathbf{c}$$

for a symmetric matrix $\mathbf{H}$. What is the Hessian in this case? What is the gradient of q when $\mathbf{d} = \mathbf{0}$? What can go wrong if the conjugate gradient method is applied to the quadratic model to get the search direction $\mathbf{d}$?

Solution: The Hessian is $2\mathbf{H}$, and

$$\nabla q(\mathbf{d}) = \left(\mathbf{H} + \mathbf{H}^\top\right)\mathbf{d} + \mathbf{b} = (2\mathbf{H})\mathbf{d} + \mathbf{b}$$

The gradient is $\mathbf{b}$ when $\mathbf{d} = \mathbf{0}$. The conjugate gradient method may diverge because $\mathbf{H}$ is not guaranteed to be positive definite.

Exercise 5.6. How is Nesterov momentum an improvement over momentum?

Solution: Nesterov momentum looks at the point where you will be after the update to compute the update itself.

Exercise 5.7. In what way is the conjugate gradient method an improvement over steepest descent?

Solution: The conjugate gradient method implicitly reuses previous information about the function and thus may enjoy better convergence in practice.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 5.8. In conjugate gradient descent, what is the descent direction at the first iteration for the function $f(x,y) = x^2 + xy + y^2 + 5$ when initialized at $(x,y) = (1,1)$? What is the resulting point after two steps of the conjugate gradient method?

Solution: The conjugate gradient method initially follows the steepest descent direction. The gradient is

$$\nabla f(x,y) = [2x + y, 2y + x]$$

which for $(x,y) = (1,1)$ is $[3,3]$. The direction of steepest descent is opposite the gradient, $\mathbf{d}^{(1)} = [-3,-3]$.

The Hessian is

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Since the function is quadratic and the Hessian is positive definite, the conjugate gradient method converges in at most two steps. Thus, the resulting point after two steps is the optimum, $(x,y) = (0,0)$, where the gradient is zero.

Exercise 5.9. We have a polynomial function f such that $f(\mathbf{x}) > 2$ for all $\mathbf{x}$ in three-dimensional Euclidean space. Suppose we are using steepest descent with step lengths optimized at each step, and we want to find a local minimum of f. If our unnormalized descent direction is $[1,2,3]$ at step k, is it possible for our unnormalized descent direction at step $k+1$ to be $[0,0,-3]$? Why or why not?

Solution: No. If exact minimization is performed, then the descent directions between steps are orthogonal, but $[1,2,3]^\top [0,0,-3] \neq 0$.

Exercise 5.10. The *heavy ball* method[17] modifies gradient descent by pushing the new iterate further in the direction of its most recent update:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \mathbf{g}^{(k)} + \beta\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)$$

[17] B. T. Polyak, "Some Methods of Speeding up the Convergence of Iteration Methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

Show that the iterates produced by the heavy ball method are identical to that of the momentum method.

Solution: Let us interpret the difference in iterates as a momentum vector:

$$\mathbf{v}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$$

The heavy ball update is then:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \mathbf{g}^{(k)} + \beta\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right) = \mathbf{x}^{(k)} - \alpha \mathbf{g}^{(k)} + \beta \mathbf{v}^{(k)}$$

Its momentum update is then:

$$\mathbf{v}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} = -\alpha \mathbf{g}^{(k)} + \beta \mathbf{v}^{(k)}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

This matches the momentum update:

$$\mathbf{v}^{(k+1)} = \beta\mathbf{v}^{(k)} - \alpha\mathbf{g}^{(k)}$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$

Thus, the two methods will produce the same iterates for the same parameters and initial conditions.

Exercise 5.11. Design parameter scale can make a significant difference when optimizing an objective function. Consider a trivial quadratic problem, minimize $f(\mathbf{x}) = x_1^2 + x_2^2$.

Manually work out the first three steps of gradient descent on the objective function where $\mathbf{x}^{(0)} = [3\,\text{m}, 3\,\text{m}]$ using $\alpha = 0.1$. Then work out the first three steps on the equivalent problem minimize $f(\mathbf{x}) = x_1^2 + 10^{-6}x_2^2$ from $\mathbf{x}^{(0)} = [3\,\text{m}, 3{,}000\,\text{mm}]$. Do the iterates proceed differently?

Solution:

The iterates for the first problem are:

$$x^{(1)} = \mathbf{x}^{(0)} - \alpha\nabla f(\mathbf{x}^{(0)}) = [3.000, 3.000] - \alpha[6.000, 6.000] = [2.400, 2.400]$$
$$x^{(2)} = \mathbf{x}^{(1)} - \alpha\nabla f(\mathbf{x}^{(1)}) = [2.400, 2.400] - \alpha[4.800, 4.800] = [1.920, 1.920]$$
$$x^{(3)} = \mathbf{x}^{(2)} - \alpha\nabla f(\mathbf{x}^{(2)}) = [1.920, 1.920] - \alpha[3.840, 3.840] \approx [1.536, 1.536]$$

The iterates for the second problem are:

$$x^{(1)} = \mathbf{x}^{(0)} - \alpha\nabla f(\mathbf{x}^{(0)}) = [3.000, 3000.0000] - \alpha[6.000, 0.006] \approx [2.400, 2999.9994]$$
$$x^{(2)} = \mathbf{x}^{(1)} - \alpha\nabla f(\mathbf{x}^{(1)}) = [2.400, 2999.9994] - \alpha[4.800, 0.006] \approx [1.920, 2999.9988]$$
$$x^{(3)} = \mathbf{x}^{(2)} - \alpha\nabla f(\mathbf{x}^{(2)}) = [1.920, 2999.9988] - \alpha[3.840, 0.006] \approx [1.536, 2999.9982]$$

We find that the poor scaling of the second problem causes descent to almost exclusively adjust the first parameter, making almost no progress in the second parameter.

Exercise 5.12. The tip deflection of a cantilever beam of length ℓ subject to a point force F at its end is $\delta = F\ell^3/3EI$, where E is the Young's modulus, a scalar measure of the stiffness of the material, and I is the area moment of inertia. For a beam with a square cross section with side lengths d, the area moment of inertia is: $I = d^4/12$.

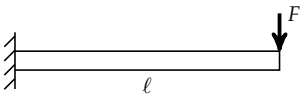


Figure 5.8. A cantilever beam of length ℓ affixed on the left side, with a point force applied at the far end.

Suppose we have such a cantilever beam with $E = 2 \times 10^{11}\,\text{N/m}^2$, a force $F = 500\,\text{N}$, and that we wish to find a beam length ℓ and side length d such that the tip deflection is $\delta^\star = 0.1\,\text{m}$. Manually work out the first three steps of gradient descent on the objective function $f(\ell, d) = (\delta(\ell, d) - \delta^\star)^2$ starting from $\ell^{(0)} = 5\,\text{m}$ and $d^{(0)} = 0.1\,\text{m}$ and using $\alpha = 0.1$. Describe any issues you observe.

Solution:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The objective function, dropping units since they are consistent, is:

$$f(\ell, d) = \left(\frac{500\ell^3}{3 \cdot 2 \times 10^{11} \frac{d^4}{12}} - 0.1\right)^2 = \left(10^{-8}\frac{\ell^3}{d^4} - 0.1\right)^2$$

The gradient is:

$$\nabla f(\ell, d) = 2\left(10^{-8}\frac{\ell^3}{d^4} - 0.1\right) \cdot 10^{-8}\left[3\frac{\ell^2}{d^4}, -4\frac{\ell^3}{d^5}\right]$$

The iterations are:

$$x^{(1)} = \mathbf{x}^{(0)} - \alpha\nabla f(\mathbf{x}^{(0)}) = [5.0000, 0.1000] - \alpha[-0.0013, 0.0875] \approx [5.0001, 0.0913]$$
$$x^{(2)} = \mathbf{x}^{(1)} - \alpha\nabla f(\mathbf{x}^{(1)}) = [5.0001, 0.0913] - \alpha[-0.0018, 0.1296] \approx [5.0003, 0.0783]$$
$$x^{(3)} = \mathbf{x}^{(2)} - \alpha\nabla f(\mathbf{x}^{(2)}) = [5.0003, 0.0783] - \alpha[-0.0027, 0.2269] \approx [5.0006, 0.0556]$$

We find that the initial design has too little deflection, so the iterates increase the length of the beam and decrease its thickness. However, the thickness changes much more rapidly than the length, which causes the beam's thickness to rapidly shrink with almost no change to the length. A larger step factor could easily cause the thickness to become negative.

Furthermore, the gradient is getting steeper as the iterates progress. The objective function is not convex, and further iterations will overshoot the optimum.

Exercise 5.13. Revisit the previous problem, but apply a log transformation to the design variables:[18]

$$\tilde{\ell} = \log(\ell) \qquad \tilde{d} = \log(d)$$

and minimize the squared error of the log of the deflection:

$$f(\tilde{\ell}, \tilde{d}) = \left(\log\delta(e^{\tilde{\ell}}, e^{\tilde{d}}) - \log\delta^\star\right)^2$$

Again, manually work out the first three steps of gradient descent from the same initial design, this time with $\alpha = 0.01$. Describe any differences with respect to the previous problem.

[18] This transformation additionally ensures that the beam length and thickness remain positive. Solutions to the new problem map to solutions to the original problem.

Solution:

The new objective function, ignoring units, is a simple quadratic:

$$\begin{aligned} f(\tilde{\ell}, \tilde{d}) &= \left(\log\delta(e^{\tilde{\ell}}, e^{\tilde{d}}) - \log\delta^\star\right)^2 \\ &= \left(\log\left(\frac{4 \cdot 500}{2 \times 10^{11}} e^{3\tilde{\ell} - 4\tilde{d}}\right) - \log 0.1\right)^2 \\ &= \left(\log\left(10^{-8} e^{3\tilde{\ell} - 4\tilde{d}}\right) - \log 0.1\right)^2 \\ &= \left(3\tilde{\ell} - 4\tilde{d} - 16.118\right)^2 \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The gradient is:

$$\nabla f(\tilde{\ell}, \tilde{d}) = 2\left(3\tilde{\ell} - 4\tilde{d} - 16.118\right)[3, -4]$$

The equivalent initial design is $\tilde{\ell}^{(0)} = \log(5) = 1.609$, $\tilde{d}^{(0)} = \log(0.1) = -2.303$.
The iterations are:

$$x^{(1)} = \mathbf{x}^{(0)} - \alpha \nabla f(\mathbf{x}^{(0)}) = [1.609, -2.303] - \alpha[-12.477, 16.636] \approx [1.734, -2.469]$$
$$x^{(2)} = \mathbf{x}^{(1)} - \alpha \nabla f(\mathbf{x}^{(1)}) = [1.734, -2.469] - \alpha[\ -6.238, 8.318] \approx [1.797, -2.552]$$
$$x^{(3)} = \mathbf{x}^{(2)} - \alpha \nabla f(\mathbf{x}^{(2)}) = [1.797, -2.552] - \alpha[\ -3.119, 4.159] \approx [1.828, -2.594]$$

The final iterate corresponds to $\ell = 6.220\,\text{m}$ and $d = 0.075\,\text{m}$. The problem transformation did a better job of balancing out the gradients with respect to the magnitudes of the two design components, and there is no risk of producing a negative thickness.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

6 Second-Order Methods

The previous chapter focused on optimization methods that involve first-order approximations of the objective function using the gradient. This chapter focuses on leveraging *second-order* approximations that use the second derivative in univariate optimization or the Hessian in multivariate optimization to direct the search. This additional information can help improve the local model used for informing the selection of directions and step lengths in descent algorithms.

6.1 Newton's Method

Knowing the function value and gradient for a design point can help determine the direction to travel, but this first-order information does not directly help determine how far to step to reach a local minimum. Second-order information, on the other hand, allows us to make a quadratic approximation of the objective function and approximate the right step size to reach a local minimum as shown in figure 6.1. As we have seen with quadratic fit search in chapter 3, we can analytically obtain the location where a quadratic approximation has a zero gradient. We can then use that location as the next iterate to approach a local minimum.

In univariate optimization, the quadratic approximation about a point $x^{(k)}$ comes from the second-order Taylor expansion:

$$q(x) = f(x^{(k)}) + (x - x^{(k)})f'(x^{(k)}) + \frac{(x - x^{(k)})^2}{2} f''(x^{(k)}) \tag{6.1}$$

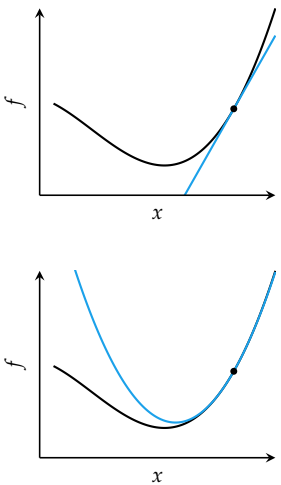


Figure 6.1. A comparison of first-order and second-order approximations. Bowl-shaped quadratic approximations have unique locations where the derivative is zero.

Setting the derivative to zero and solving for the root yields the update equation for *Newton's method*:

$$\frac{\partial}{\partial x} q(x) = f'(x^{(k)}) + (x - x^{(k)}) f''(x^{(k)}) = 0 \tag{6.2}$$

$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} \tag{6.3}$$

This update is shown in figure 6.2.

The update rule in Newton's method involves dividing by the second derivative. The update is undefined if the second derivative is zero, which occurs when the quadratic approximation is a line. Instability also occurs when the second derivative is very close to zero, in which case the next iterate will lie very far from the current design point, far from where the local quadratic approximation is valid. Poor local approximations can lead to poor performance with Newton's method. Figure 6.3 shows three kinds of failure cases.

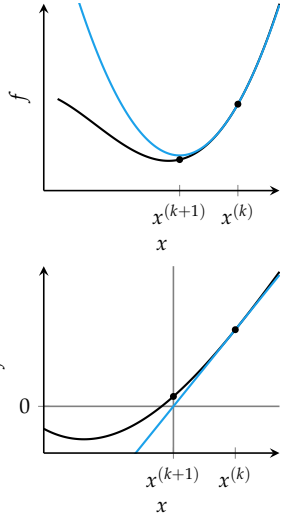


Figure 6.2. Newton's method can be interpreted as a root-finding method applied to f' that iteratively improves a univariate design point by taking the tangent line at $(x, f'(x))$, finding the intersection with the x-axis, and using that x value as the next design point.

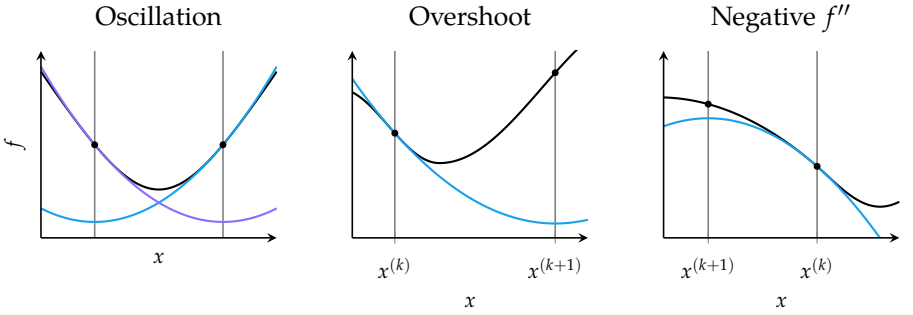


Figure 6.3. Examples of failure cases with Newton's method.

Newton's method does tend to converge quickly when in a bowl-like region that is sufficiently close to a local minimum. It has *quadratic convergence*, meaning the difference between the minimizer and the iterate is approximately squared with every iteration. This rate of convergence holds for Newton's method starting from $x^{(1)}$ within a distance $\delta \ll 1$ of a root x^* if for all x in the interval $[x^* - \delta, x^* + \delta]$:

- $f''(x) \neq 0$,
- $f'''(x)$ is continuous, and
- $\frac{1}{2}\left|\frac{f'''(x^{(1)})}{f''(x^{(1)})}\right| < c\left|\frac{f'''(x^*)}{f''(x^*)}\right|$ for some $c < \infty$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The final condition guards against overshoot.[1]

Newton's method can be extended to multivariate optimization (algorithm 6.1). The multivariate second-order Taylor expansion at $\mathbf{x}^{(k)}$ is the quadratic:

$$q(\mathbf{x}) = f(\mathbf{x}^{(k)}) + (\mathbf{g}^{(k)})^\top(\mathbf{x} - \mathbf{x}^{(k)}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(k)})^\top \mathbf{H}^{(k)}(\mathbf{x} - \mathbf{x}^{(k)}) \tag{6.4}$$

where $\mathbf{g}^{(k)}$ and $\mathbf{H}^{(k)}$ are the gradient and Hessian at $\mathbf{x}^{(k)}$, respectively.

We evaluate the gradient and set it to zero:

$$\nabla q(\mathbf{x}) = \mathbf{g}^{(k)} + \mathbf{H}^{(k)}(\mathbf{x} - \mathbf{x}^{(k)}) = \mathbf{0} \tag{6.5}$$

We then solve for the next iterate, thereby obtaining Newton's method in multivariate form:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (\mathbf{H}^{(k)})^{-1}\mathbf{g}^{(k)} \tag{6.6}$$

```
struct NewtonsMethod <: DescentMethod end
step!(M::NewtonsMethod, f, ∇f, H, x) = x - H(x) \ ∇f(x)
```

Algorithm 6.1. Newton's method, which uses a second-order approximation at `x` based on the Hessian of the objective function `H` and the objective function gradient `∇f`. The objective function input is not used.

If f is quadratic and its Hessian is positive definite, then the update converges to the global minimum in one step. For general functions, Newton's method is often terminated once x ceases to change by more than a given tolerance.[2] Example 6.1 shows how Newton's method can be used to minimize a function.

Newton's method can also be used to supply a descent direction to line search or can be modified to use a step factor.[3] Smaller steps toward the minimum or line searches along the descent direction can increase the method's robustness. The descent direction is:[4]

$$\mathbf{d}^{(k)} = -(\mathbf{H}^{(k)})^{-1}\mathbf{g}^{(k)} \tag{6.7}$$

6.2 Secant Method

Newton's method for univariate function minimization requires the first and second derivatives f' and f''. In many cases, f' is known but the second derivative is not. The *secant method* (algorithm 6.2) applies Newton's method using estimates of the second derivative and thus only requires f'. This property makes the secant method more convenient to use in practice.

[1] The final condition enforces *sufficient closeness*, ensuring that the function is sufficiently approximated by the Taylor expansion. J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 3rd ed. Springer, 2002.

[2] Termination conditions for descent methods are given in chapter 4.

[3] See chapter 4.

[4] The descent direction given by Newton's method is similar to the *natural gradient* or *covariant gradient*. S. Amari, "Natural Gradient Works Efficiently in Learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 6.1. Newton's method used to minimize Booth's function; see appendix B.2.

With $\mathbf{x}^{(1)} = [9, 8]$, we will use Newton's method to minimize Booth's function:

$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

The gradient of Booth's function is:

$$\nabla f(\mathbf{x}) = [10x_1 + 8x_2 - 34, 8x_1 + 10x_2 - 38]$$

The Hessian of Booth's function is:

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}$$

The first iteration of Newton's method yields:

$$\begin{aligned} \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} - \left(\mathbf{H}^{(1)}\right)^{-1} \mathbf{g}^{(1)} = \begin{bmatrix} 9 \\ 8 \end{bmatrix} - \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}^{-1} \begin{bmatrix} 10 \cdot 9 + 8 \cdot 8 - 34 \\ 8 \cdot 9 + 10 \cdot 8 - 38 \end{bmatrix} \\ &= \begin{bmatrix} 9 \\ 8 \end{bmatrix} - \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}^{-1} \begin{bmatrix} 120 \\ 114 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \end{aligned}$$

The gradient at $\mathbf{x}^{(2)}$ is zero, so we have converged after a single iteration. The Hessian is positive definite everywhere, so $\mathbf{x}^{(2)}$ is the global minimum.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The secant method uses the last two iterates to approximate the second derivative:

$$f''(x^{(k)}) \approx \frac{f'(x^{(k)}) - f'(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \tag{6.8}$$

This estimate is substituted into Newton's method:

$$x^{(k+1)} \leftarrow x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f'(x^{(k)}) - f'(x^{(k-1)})} f'(x^{(k)}) \tag{6.9}$$

The secant method requires an additional initial design point. It suffers from the same problems as Newton's method and may take more iterations to converge due to approximating the second derivative.

```
mutable struct SecantMethod <: DescentMethod
    α      # step factor, used in initialization
    x_prev # previous design
    g_prev # previous gradient
end
function init!(M::SecantMethod, f, f′, x)
    α = M.α
    M.x_prev = x + α*f′(x)
    M.g_prev = f′(M.x_prev)
    return M
end
function step!(M::SecantMethod, f, f′, x)
    x_prev, g_prev = M.x_prev, M.g_prev
    g = f′(x)
    x′ = x - (x - x_prev)/(g - g_prev)*g
    M.x_prev, M.g_prev = x, g
    return x′
end
```

Algorithm 6.2. The secant method for minimizing a univariate objective function `f` using only the first derivative `f′` and previous values. This init method uses a small gradient step with a step factor `α` in order to initialize a previous value.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

6.3 Levenberg-Marquardt Algorithm

The *Levenberg-Marquardt algorithm*[5] (algorithm 6.3) automatically interpolates between approximate Newton updates and gradient descent steps. Newton's method tends to perform well when a quadratic approximation is a good fit for our objective function, but will perform poorly when the fit is poor. A quadratic approximation for a smooth function will tend to be good sufficiently close to a local optimum, where the objective function is typically convex and bowl-like. In more linear regions, quadratic approximations will be poor, and in concave regions, quadratic approximations will have degenerate Hessians. In such cases it is often better to simply use a gradient descent step. These regimes are shown in figure 6.4.

The interpolating update rule is parameterized by a damping factor δ:

$$\mathbf{x}' = \mathbf{x} - (\mathbf{H} + \delta\mathbf{I})^{-1}\mathbf{g} \tag{6.10}$$

When δ is small, the update mimics Newton's method. When δ is large, the update mimics gradient descent with a step factor $\alpha \approx 1/\delta$.

We adjust the damping factor during descent based on whether our iterates improve the objective function value. In every iteration, the gradient and the approximate Hessian are calculated and used to evaluate a candidate next iterate. If the objective is better at the next iterate, it is accepted, and δ can be decreased. If the objective is worse at the next iterate, it is rejected, the algorithm retains the current iterate, and δ is increased.

The Levenberg-Marquardt update rule incorporates a slight adjustment to equation (6.10):[6]

$$\mathbf{x}' = \mathbf{x} - (\mathbf{H} + \delta\text{diag}(\text{diag}(\mathbf{H})))^{-1}\mathbf{g} \tag{6.11}$$

which incorporates the diagonal of the Hessian. This leverages information about the Hessian even when mimicking gradient descent, allowing iterates to move further in directions where the curvature is smaller.

If the Hessian is not invertible and $\text{diag}(\mathbf{H})$ has any zero entries, increasing the damping factor will not produce an invertible matrix. To mitigate this effect, we can take the component-wise maximum of these values with a small positive number, as done in the implementation. Figure 6.5 shows the Levenberg-Marquardt method on a test function.

[5] This algorithm is named for the American statisticians Kenneth Levenberg (1919–1973) and Donald Marquardt (1929–1997). It was originally published by Levenberg while working for the Frankford Army Arsenal. K. Levenberg, "A Method for the Solution of Certain Non-Linear Problems in Least Squares," *Quarterly of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, 1944. It was later rediscovered by Marquardt while working for DuPont. D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963. There were several other independent rediscoveries of this general method.

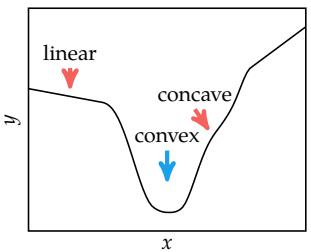


Figure 6.4. The local curvature of the optimization function will determine the efficacy of Newton's method.

[6] If the input to diag is a matrix, diag extracts the diagonal of the input. When the input is a vector, it returns a diagonal matrix with the input as the diagonal. Hence, $\text{diag}(\text{diag}(\mathbf{H}))$ produces a diagonal matrix whose diagonal matches that of $\mathbf{H}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function levenberg_marquardt(f, ∇f, H, x, δ, γ_acc, γ_rej, ϵ)
    M = H(x)
    d = max.(diag(M), ϵ)
    M += δ*Diagonal(d)
    x′ = x - M \ ∇f(x)
    if f(x′) < f(x)
        # accept the new design and decrease damping
        return (x=x′, δ=δ*γ_acc)
    end
    # reject the new design and increase damping
    return (x=x, δ=δ*γ_rej)
end
```

Algorithm 6.3. The Levenberg-Marquardt algorithm for interpolating between standard gradient descent and Newton steps. It takes an objective function `f`, a gradient `∇f`, and a Hessian `H`, which are evaluated at a design `x`. The interpolation is controlled by a damping factor `δ > 0`. This damping factor is increased or decreased by scaling with `γ_acc` or `γ_rej` based on whether the next iterate is an improvement. Taking a component-wise max with a small positive scalar `ϵ` ensures that the matrix can become invertible for a sufficiently large damping factor.

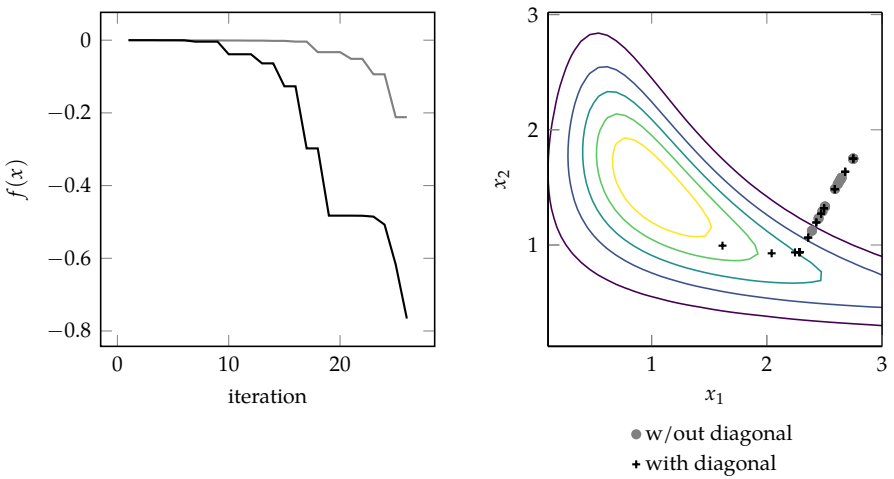


Figure 6.5. The Levenberg-Marquardt method run on Wheeler's Ridge (appendix B.7) using the update with the diagonal of the Hessian in equation (6.11), compared against the update without the diagonal given in equation (6.10). The methods start in the same location in a relatively flat region, and they must traverse that region to a more bowl-like region containing the minimum. We set `γ_acc=0.1`, `γ_rej=10.0`, and `ϵ=1e-6`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

6.4 Levenberg-Marquardt for Sum of Squares

The Levenberg-Marquardt algorithm was originally developed for problems involving sums of squares, for which we can derive an efficient *outer product approximation* of the Hessian. Consider an objective function:

$$f(\mathbf{x}) = \sum_i f_i(\mathbf{x})^2 \tag{6.12}$$

The gradient of this objective is:

$$\nabla f(\mathbf{x}) = 2\sum_i f_i(\mathbf{x})\nabla f_i(\mathbf{x}) \tag{6.13}$$

We can form a linear approximation for each of the constituent objective functions at our current iterate $\mathbf{x}^{(k)}$:

$$f_i(\mathbf{x}) \approx f_i(\mathbf{x}^{(k)}) + \nabla f_i(\mathbf{x}^{(k)})^\top \left(\mathbf{x} - \mathbf{x}^{(k)}\right) \tag{6.14}$$

Substituting them into the gradient of the overall objective yields[7]

$$\nabla f(\mathbf{x}) \approx 2\sum_i \left(f_i(\mathbf{x}^{(k)}) + \nabla f_i(\mathbf{x}^{(k)})^\top (\mathbf{x} - \mathbf{x}^{(k)})\right)\nabla f_i(\mathbf{x}) \tag{6.15}$$

$$= 2\sum_i f_i(\mathbf{x}^{(k)})\nabla f_i(\mathbf{x}^{(k)}) + 2\sum_i \nabla f_i(\mathbf{x}^{(k)})\nabla f_i(\mathbf{x}^{(k)})^\top (\mathbf{x} - \mathbf{x}^{(k)}) \tag{6.16}$$

$$= 2\tilde{\mathbf{g}}^{(k)} + 2\tilde{\mathbf{H}}^{(k)}(\mathbf{x} - \mathbf{x}^{(k)}) \tag{6.17}$$

where we define

$$\tilde{\mathbf{g}}^{(k)} = \sum_i f_i(\mathbf{x}^{(k)})\nabla f_i(\mathbf{x}^{(k)}) \tag{6.18}$$

$$\tilde{\mathbf{H}}^{(k)} = \sum_i \nabla f_i(\mathbf{x}^{(k)})\nabla f_i(\mathbf{x}^{(k)})^\top \tag{6.19}$$

Setting this gradient to zero and solving for $\mathbf{x}$ yields the new iterate:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\tilde{\mathbf{H}}^{(k)}\right)^{-1} \tilde{\mathbf{g}}^{(k)} \tag{6.20}$$

This update matches Newton's method in equation (6.6), except the Hessian is approximated using the outer product of the gradients. Incorporating the damping factor produces the Levenberg-Marquardt update (equation (6.11)):

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\tilde{\mathbf{H}}^{(k)} + \delta \text{diag}(\text{diag}(\tilde{\mathbf{H}}^{(k)}))\right)^{-1} \tilde{\mathbf{g}}^{(k)} \tag{6.21}$$

The estimates of the gradient and Hessian are implemented in algorithm 6.4.

[7] This is because the gradient is equal everywhere in a first-order approximation:

$$\nabla f_i(\mathbf{x}) = \nabla f_i(\mathbf{x}^{(k)})$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function sum_of_squares_descent_estimates(fs, ∇fs, x)
    g = sum(f(x)*∇f(x) for (f,∇f) in zip(fs, ∇fs))
    H = sum(∇f(x)*∇f(x)' for ∇f in ∇fs)
    return (g, H)
end
```

Algorithm 6.4. An algorithm for computing the gradient and outer product Hessian approximation for a sum-of-squares optimization problem. It takes as input a list of objective functions `fs`, their gradients `∇fs`, and a design `x`.

6.5 Quasi-Newton Methods

Just as the secant method approximates f'' in the univariate case, *quasi-Newton* methods approximate the inverse Hessian. Quasi-Newton method updates have the form:

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)}\mathbf{Q}^{(k)}\mathbf{g}^{(k)} \tag{6.22}$$

where $\alpha^{(k)}$ is a scalar step factor and $\mathbf{Q}^{(k)}$ approximates the inverse of the Hessian at $\mathbf{x}^{(k)}$.

These methods typically set $\mathbf{Q}^{(1)}$ to the identity matrix, and they then apply updates to reflect information learned with each iteration. To simplify the equations for the various quasi-Newton methods, we define the following:

$$\boldsymbol{\gamma}^{(k+1)} \equiv \mathbf{g}^{(k+1)} - \mathbf{g}^{(k)} \tag{6.23}$$

$$\boldsymbol{\delta}^{(k+1)} \equiv \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \tag{6.24}$$

The *Davidon-Fletcher-Powell* (DFP) method (algorithm 6.5) uses:[8]

$$\mathbf{Q} \leftarrow \mathbf{Q} - \frac{\mathbf{Q}\boldsymbol{\gamma}\boldsymbol{\gamma}^\top\mathbf{Q}}{\boldsymbol{\gamma}^\top\mathbf{Q}\boldsymbol{\gamma}} + \frac{\boldsymbol{\delta}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top\boldsymbol{\gamma}} \tag{6.25}$$

where all terms on the right hand side are evaluated at the same iteration.

The update for $\mathbf{Q}$ in the DFP method keeps $\mathbf{Q}$ symmetric and positive definite. If $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{A}\mathbf{x} + \mathbf{b}^\top\mathbf{x} + c$, then the inverse Hessian is $\mathbf{A}^{-1}$, which DFP approximates iteratively. For such quadratic functions, DFP converges in n steps with exact line searches, similar to the conjugate gradient method. For high-dimensional problems, storing and updating $\mathbf{Q}$ can be significant compared to other methods like the conjugate gradient method.

An alternative to DFP, the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method (algorithm 6.6), uses:[9]

$$\mathbf{Q} \leftarrow \mathbf{Q} - \left(\frac{\boldsymbol{\delta}\boldsymbol{\gamma}^\top\mathbf{Q} + \mathbf{Q}\boldsymbol{\gamma}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top\boldsymbol{\gamma}}\right) + \left(1 + \frac{\boldsymbol{\gamma}^\top\mathbf{Q}\boldsymbol{\gamma}}{\boldsymbol{\delta}^\top\boldsymbol{\gamma}}\right)\frac{\boldsymbol{\delta}\boldsymbol{\delta}^\top}{\boldsymbol{\delta}^\top\boldsymbol{\gamma}} \tag{6.26}$$

[8] The original concept was presented in a technical report, W. C. Davidon, "Variable Metric Method for Minimization," Argonne National Laboratory, Tech. Rep. ANL-5990, 1959. It was later published: W. C. Davidon, "Variable Metric Method for Minimization," *SIAM Journal on Optimization*, vol. 1, no. 1, pp. 1–17, 1991. The method was modified by R. Fletcher and M. J. D. Powell, "A Rapidly Convergent Descent Method for Minimization," *The Computer Journal*, vol. 6, no. 2, pp. 163–168, 1963.

[9] R. Fletcher, *Practical Methods of Optimization*, 2nd ed. Wiley, 1987.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct DFP <: DescentMethod
    Q # approximate inverse Hessian
end
function init!(M::DFP, f, ∇f, x)
    m = length(x)
    M.Q = Matrix(1.0I(m))
    return M
end
function step!(M::DFP, f, ∇f, x)
    Q, g = M.Q, ∇f(x)
    x′ = line_search(f, x, -Q*g)
    g′ = ∇f(x′)
    δ = x′ - x
    γ = g′ - g
    Q .= Q - Q*γ*γ'*Q/(γ'*Q*γ) + δ*δ'/(δ'*γ)
    return x′
end
```

Algorithm 6.5. The Davidon-Fletcher-Powell descent method.

```
mutable struct BFGS <: DescentMethod
    Q # approximate inverse Hessian
end
function init!(M::BFGS, f, ∇f, x)
    m = length(x)
    M.Q = Matrix(1.0I(m))
    return M
end
function step!(M::BFGS, f, ∇f, x)
    Q, g = M.Q, ∇f(x)
    x′ = line_search(f, x, -Q*g)
    g′ = ∇f(x′)
    δ = x′ - x
    γ = g′ - g
    Q .= Q - (δ*γ'*Q + Q*γ*δ')/(δ'*γ) +
              (1 + (γ'*Q*γ)/(δ'*γ))[1]*(δ*δ')/(δ'*γ)
    return x′
end
```

Algorithm 6.6. The Broyden-Fletcher-Goldfarb-Shanno descent method.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

BFGS does better than DFP with approximate line search but still uses an $n \times n$ dense matrix. For very large problems where space is a concern, the *Limited-memory BFGS* method (algorithm 6.7), or *L-BFGS*, can be used to approximate BFGS.[10] L-BFGS stores the last m values for $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ rather than the full inverse Hessian, where $i = 1$ indexes the oldest value and $i = m$ indexes the most recent.

[10] J. Nocedal, "Updating Quasi-Newton Matrices with Limited Storage," *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980.

The process for computing the descent direction $\mathbf{d}$ at $\mathbf{x}$ begins by computing $\mathbf{q}^{(m)} = \nabla f(\mathbf{x})$. The backward pass computes vectors $\mathbf{q}^{(i-1)}$ for i from m down to 1 using the stored history:

$$\begin{aligned} \alpha^{(i)} &= \frac{\left(\boldsymbol{\delta}^{(i)}\right)^\top \mathbf{q}^{(i)}}{\left(\boldsymbol{\delta}^{(i)}\right)^\top \boldsymbol{\gamma}^{(i)}} \\ \mathbf{q}^{(i-1)} &= \mathbf{q}^{(i)} - \alpha^{(i)} \boldsymbol{\gamma}^{(i)} \end{aligned} \tag{6.27}$$

These vectors are used to compute another $m + 1$ vectors, starting by scaling the result of the backward pass $\mathbf{q}^{(0)}$:

$$\mathbf{z}^{(0)} = \frac{\left(\boldsymbol{\delta}^{(m)}\right)^\top \boldsymbol{\gamma}^{(m)}}{\left(\boldsymbol{\gamma}^{(m)}\right)^\top \boldsymbol{\gamma}^{(m)}} \mathbf{q}^{(0)} \tag{6.28}$$

The forward pass proceeds with $\mathbf{z}^{(i)}$ for i from 1 to m:

$$\mathbf{z}^{(i)} = \mathbf{z}^{(i-1)} + \boldsymbol{\delta}^{(i)} \left(\alpha^{(i)} - \frac{\left(\boldsymbol{\gamma}^{(i)}\right)^\top \mathbf{z}^{(i-1)}}{\left(\boldsymbol{\delta}^{(i)}\right)^\top \boldsymbol{\gamma}^{(i)}} \right) \tag{6.29}$$

The descent direction is $\mathbf{d} = -\mathbf{z}^{(m)}$.

The initial inverse Hessian approximation $\mathbf{Q}$ is implicitly set to a scaled identity matrix. The scaling factor uses the most recent step information:

$$\mathbf{Q} = \frac{\left(\boldsymbol{\delta}^{(m)}\right)^\top \boldsymbol{\gamma}^{(m)}}{\left(\boldsymbol{\gamma}^{(m)}\right)^\top \boldsymbol{\gamma}^{(m)}} \mathbf{I} \tag{6.30}$$

Substituting this into $\mathbf{z}^{(0)} = \mathbf{Q}\mathbf{q}^{(0)}$ initializes the forward pass.

The quasi-Newton methods discussed in this section are compared in figure 6.6. They often perform quite similarly.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

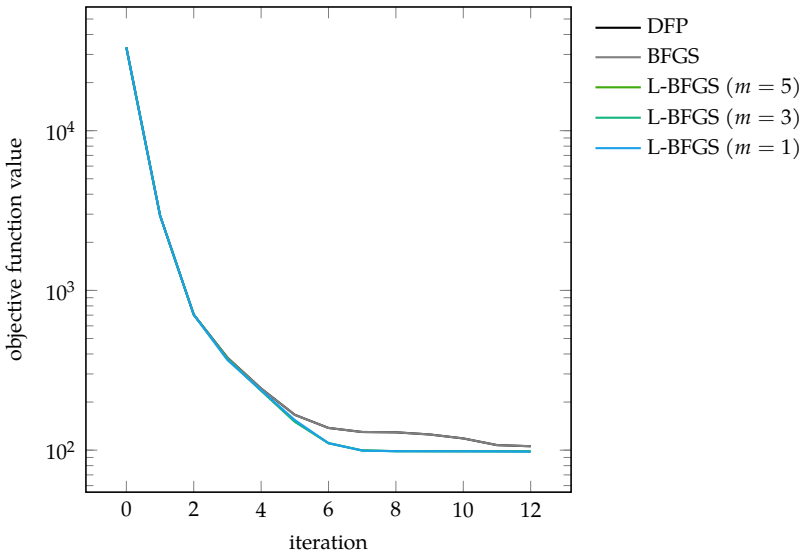


Figure 6.6. Several quasi-Newton methods compared on a 100-dimensional extension of the Rosenbrock function; see algorithm B.7. The limited-memory BFGS method underperforms relative to the other methods.

6.6 Summary

- Incorporating second-order information in descent methods often speeds convergence.
- Newton's method is a root-finding method that leverages second-order information to quickly descend to a local minimum.
- The secant method and quasi-Newton methods approximate Newton's method when the second-order information is not directly available.

6.7 Exercises

Exercise 6.1. What advantage does second-order information provide about convergence that first-order information lacks?

Solution: Second-order information can guarantee that one is at a local minimum, whereas a gradient of zero is necessary but insufficient to guarantee local optimality.

Exercise 6.2. When finding roots in one dimension, when would we use Newton's method instead of the bisection method?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct LimitedMemoryBFGS <: DescentMethod
    m  # history size
    δs # step differences
    γs # gradient changes
    αs # inverse Hessian contributions
end
function init!(M::LimitedMemoryBFGS, f, ∇f, x)
    M.δs = []
    M.γs = []
    M.αs = []
    return M
end
function step!(M::LimitedMemoryBFGS, f, ∇f, x)
    δs, γs, αs, g = M.δs, M.γs, M.αs, ∇f(x)
    m = length(δs)
    if m > 0
        q = g
        for i in m:-1:1
            αs[i] = (δs[i]⋅q)/(δs[i]⋅γs[i])
            q -= αs[i]*γs[i]
        end
        z = (δs[m]⋅γs[m])/(γs[m]⋅γs[m])*q
        for i in 1:m
            z += δs[i]*(αs[i] - (γs[i]⋅z)/(δs[i]⋅γs[i]))
        end
        x′ = line_search(f, x, -z)
    else
        x′ = line_search(f, x, -g)
    end
    g′ = ∇f(x′)
    push!(δs, x′ - x); push!(γs, g′ - g)
    push!(αs, 0.0)
    while length(δs) > M.m
        popfirst!(δs); popfirst!(γs); popfirst!(αs)
    end
    return x′
end
```

Algorithm 6.7. The Limited-memory BFGS descent method, which avoids storing the approximate inverse Hessian. The parameter `m` determines the history size. The `LimitedMemoryBFGS` type also stores the step differences `δs`, the gradient changes `γs`, and inverse Hessian contributions `αs`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: We would prefer Newton's method if we start sufficiently close to the root and can compute derivatives analytically. Newton's method enjoys a better rate of convergence.

Exercise 6.3. Apply Newton's method to $f(x) = x^2$ from a starting point of your choice. How many steps do we need to converge?

Solution: $f'(x) = 2x$, $f''(x) = 2$. Thus, $x^{(2)} = x^{(1)} - 2x^{(1)}/2 = 0$; that is, you converge in one step from any starting point.

Exercise 6.4. Apply Newton's method to $f(x) = \frac{1}{2}\mathbf{x}^\top \mathbf{H}\mathbf{x}$ starting from $\mathbf{x}^{(1)} = [1, 1]$. What have you observed? Use $\mathbf{H}$ as follows:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 0 & 1000 \end{bmatrix}$$

Next, apply gradient descent to the same optimization problem by stepping with the unnormalized gradient. Do two steps of the algorithm. What have you observed? Finally, apply the conjugate gradient method. How many steps do you need to converge?

Solution: Since $\nabla f(\mathbf{x}) = \mathbf{H}\mathbf{x}$, $\nabla^2 f(\mathbf{x}) = \mathbf{H}$, and $\mathbf{H}$ is nonsingular, it follows that $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} - \mathbf{H}^{-1}\mathbf{H}\mathbf{x}^{(1)} = \mathbf{0}$. That is, Newton's method converges in one step.

Gradient descent diverges:

$$\mathbf{x}^{(2)} = [1, 1] - [1, 1000] = [0, -999]$$
$$\mathbf{x}^{(3)} = [0, -999] - [0, -1000 \cdot 999] = [0, 998001]$$

Conjugate gradient descent uses the same initial search direction as gradient descent and converges to the minimum in the second step because the optimization objective is quadratic.

Exercise 6.5. Compare Newton's method and the secant method on $f(x) = x^2 + x^4$, with $x^{(1)} = -3$ and $x^{(0)} = -4$. Run each method for 10 iterations. Make two plots:

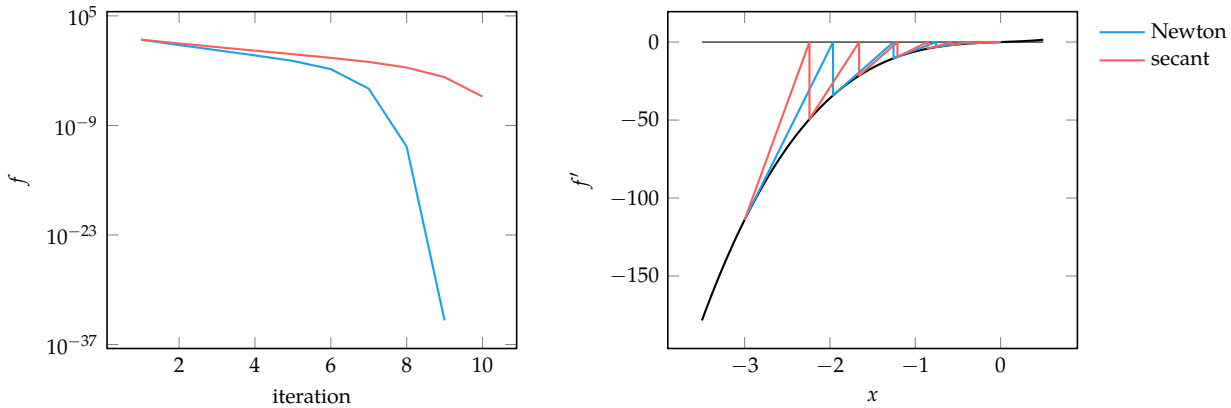
1. Plot f vs. the iteration for each method.
2. Plot f' vs. x. Overlay the progression of each method, drawing lines from $(x^{(i)}, f'(x^{(i)}))$ to $(x^{(i+1)}, 0)$ to $(x^{(i+1)}, f'(x^{(i+1)}))$ for each transition.

What can we conclude about this comparison?

Solution: The left plot shows convergence for Newton's method approaching floating-point resolution within nine iterations. The secant method is slower to converge because it can merely approximate the derivative.

The right plot shows the projection of the exact and approximate tangent lines with respect to f' for each method. The secant method's tangent lines have a higher slope, and thus intersect the x-axis prematurely.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 6.6. Give an example of a sequence of points $x^{(1)}, x^{(2)}, \ldots$ and a function f such that $f(x^{(1)}) > f(x^{(2)}) > \cdots$ and yet the sequence does not converge to a local minimum. Assume f is bounded from below.

Solution: Consider the sequence $x^{(k+1)} = x^{(k)}/2$ starting from $x^{(1)} = -1$ on the function $f(x) = x^2 - x$. Clearly the sequence converges to $x = 0$, the values for $f(x)$ are decreasing, and yet the sequence does not converge to a minimizer.

Exercise 6.7. What is the advantage of a Quasi-Newton method over Newton's method?

Solution: It does not need computation or knowledge of the entries of the Hessian, and hence does not require solving a linear system at each iteration.

Exercise 6.8. Give an example where the BFGS update does not exist. What would you do in this case?

Solution: The BFGS update does not exist when $\delta^\top \gamma \approx 0$. In that case, simply skip the update.

Exercise 6.9. Suppose we have a function $f(\mathbf{x}) = (x_1 + 1)^2 + (x_2 + 3)^2 + 4$. If we start at the origin, what is the resulting point after one step of Newton's method?

Solution: The objective function is quadratic and can thus be minimized in one step. The gradient is $\nabla f = [2(x_1 + 1), 2(x_2 + 3)]$, which is zero at $\mathbf{x}^* = [-1, -3]$. The Hessian is positive definite, so $\mathbf{x}^*$ is the minimum.

Exercise 6.10. Is the outer product approximation of the Hessian given in equation (6.19) always invertible?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: No, the outer product approximation is not necessarily invertible. Consider an outer product formed with $\mathbf{x} = [x_1, x_2]$:

$$\mathbf{H} = \mathbf{x}\mathbf{x}^\top = \begin{bmatrix} x_1^2 & x_1 x_2 \\ x_1 x_2 & x_2^2 \end{bmatrix}$$

This matrix has a determinant $x_1^2 x_2^2 - x_1 x_2 x_1 x_2 = 0$, and is thus singular.

Exercise 6.11. In this problem we will derive the optimization problem from which the Davidon-Fletcher-Powell update is obtained. Start with a quadratic approximation at $\mathbf{x}^{(k)}$:

$$f^{(k)}(\mathbf{x}) = y^{(k)} + \left(\mathbf{g}^{(k)}\right)^\top \left(\mathbf{x} - \mathbf{x}^{(k)}\right) + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}^{(k)}\right)^\top \mathbf{H}^{(k)} \left(\mathbf{x} - \mathbf{x}^{(k)}\right)$$

where $y^{(k)}$, $\mathbf{g}^{(k)}$, and $\mathbf{H}^{(k)}$ are the objective function value, the true gradient, and a positive definite Hessian approximation at $\mathbf{x}^{(k)}$.

The next iterate is chosen using line search to obtain:

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \alpha^{(k)} \left(\mathbf{H}^{(k)}\right)^{-1} \mathbf{g}^{(k)}$$

We can construct a new quadratic approximation $f^{(k+1)}$ at $\mathbf{x}^{(k+1)}$. The approximation should enforce that the local function evaluation is correct:

$$f^{(k+1)}(\mathbf{x}^{(k+1)}) = y^{(k+1)}$$

and that the local gradient is correct:

$$\nabla f^{(k+1)}(\mathbf{x}^{(k+1)}) = \mathbf{g}^{(k+1)}$$

and that the previous gradient is correct:

$$\nabla f^{(k+1)}(\mathbf{x}^{(k)}) = \mathbf{g}^{(k)}$$

Show that updating the Hessian approximation to obtain $\mathbf{H}^{(k+1)}$ requires:[11]

$$\mathbf{H}^{(k+1)} \boldsymbol{\delta}^{(k+1)} = \boldsymbol{\gamma}^{(k+1)}$$

Then, show that in order for $\mathbf{H}^{(k+1)}$ to be positive definite, we require:[12]

$$\left(\boldsymbol{\delta}^{(k+1)}\right)^\top \boldsymbol{\gamma}^{(k+1)} > 0$$

Finally, assuming that the curvature condition is enforced, explain why one then solves the following optimization problem to obtain $\mathbf{H}^{(k+1)}$:[13]

$$\begin{aligned} \underset{\mathbf{H}}{\text{minimize}} \quad & \|\mathbf{H} - \mathbf{H}^{(k)}\| \\ \text{subject to} \quad & \mathbf{H} = \mathbf{H}^\top \\ & \mathbf{H}\boldsymbol{\delta}^{(k+1)} = \boldsymbol{\gamma}^{(k+1)} \end{aligned}$$

where $\|\mathbf{H} - \mathbf{H}^{(k)}\|$ is a *matrix norm* that defines a distance between $\mathbf{H}$ and $\mathbf{H}^{(k)}$.

[11] This condition is called the *secant equation*. The vectors $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ are defined in equation (6.23).

[12] This condition is called the *curvature condition*. It can be enforced using the Wolfe conditions during line search.

[13] The Davidon-Fletcher-Powell update is obtained by solving such an optimization problem to obtain an analytical solution and then finding the corresponding update equation for the inverse Hessian approximation.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The new approximation has the form

$$f^{(k+1)}(\mathbf{x}) = y^{(k+1)} + \left(\mathbf{g}^{(k+1)}\right)^\top \left(\mathbf{x} - \mathbf{x}^{(k+1)}\right) + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}^{(k+1)}\right)^\top \mathbf{H}^{(k+1)} \left(\mathbf{x} - \mathbf{x}^{(k+1)}\right)$$

using the true function value and gradient at $\mathbf{x}^{(k+1)}$ but requires an updated Hessian $\mathbf{H}^{(k+1)}$. This form automatically satisfies $f^{(k+1)}(\mathbf{x}^{(k+1)}) = y^{(k+1)}$ and $\nabla f^{(k+1)}(\mathbf{x}^{(k+1)}) = \mathbf{g}^{(k+1)}$. We must select the new Hessian to satisfy the third condition:

$$\begin{aligned}
\nabla f^{(k+1)}(\mathbf{x}^{(k)}) &= \mathbf{g}^{(k+1)} + \mathbf{H}^{(k+1)}\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k+1)}\right) \\
&= \mathbf{g}^{(k+1)} - \mathbf{H}^{(k+1)}\left(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right) \\
&= \mathbf{g}^{(k+1)} - \mathbf{H}^{(k+1)}\boldsymbol{\delta}^{(k+1)} \\
&= \mathbf{g}^{(k)}
\end{aligned}$$

We can rearrange and substitute to obtain:

$$\mathbf{H}^{(k+1)}\boldsymbol{\delta}^{(k+1)} = \boldsymbol{\gamma}^{(k+1)}$$

Recall that a matrix $\mathbf{A}$ is positive definite if for every nonzero vector $\mathbf{x}$, $\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$. If we multiply the secant equation by $\boldsymbol{\delta}^{(k+1)}$ we obtain the curvature condition:

$$\left(\boldsymbol{\delta}^{(k+1)}\right)^\top \mathbf{H}^{(k+1)}\boldsymbol{\delta}^{(k+1)} = \left(\boldsymbol{\delta}^{(k+1)}\right)^\top \boldsymbol{\gamma}^{(k+1)} > 0$$

We seek a new positive definite matrix $\mathbf{H}^{(k+1)}$. All positive definite matrices are symmetric, so calculating a new positive definite matrix requires specifying $n(n+1)/2$ variables. The secant equation imposes n conditions on these variables, leading to an infinite number of solutions. In order to have a unique solution, we choose the positive definite matrix closest to $\mathbf{H}^{(k)}$. This objective leads to the desired optimization problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

7 Direct Methods

Direct methods rely solely on the objective function f. These methods are also called *zero-order*, *black box*, *pattern search*, or *derivative-free* methods. Direct methods do not rely on derivative information to guide them toward a local minimum or identify when they have reached a local minimum. They use other criteria to choose the next search direction and to judge when they have converged.

7.1 Cyclic Coordinate Search

Cyclic coordinate search, also known as *coordinate descent* or *taxicab search*, simply alternates between coordinate directions for its line search. The search starts from an initial $\mathbf{x}^{(1)}$ and optimizes the first input:

$$\mathbf{x}_1^{(2)} = \arg\min_{x_1} f(x_1, x_2^{(1)}, x_3^{(1)}, \ldots, x_n^{(1)}) \tag{7.1}$$

Having solved this, it optimizes the next coordinate:

$$\mathbf{x}_2^{(3)} = \arg\min_{x_2} f(x_1^{(2)}, x_2, x_3^{(2)}, \ldots, x_n^{(2)}) \tag{7.2}$$

This process is equivalent to doing a sequence of line searches along the set of n basis vectors, where the ith basis vector is all zero except for the ith component, which has value 1 (algorithm 7.1). For example, the third basis vector in a four-dimensional space is $\mathbf{e}^{(3)} = [0, 0, 1, 0]$. Figure 7.1 shows an example of a search through a two-dimensional space.

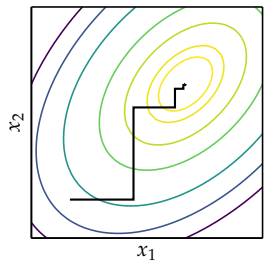


Figure 7.1. Cyclic coordinate descent alternates between coordinate directions.

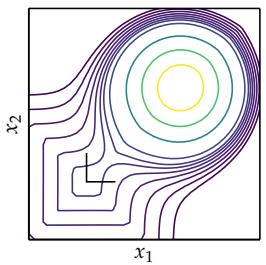


Figure 7.2. Above is an example of how cyclic coordinate search can get stuck. Moving in either of the coordinate directions will result only in increasing f, but moving diagonally, which is not allowed in cyclic coordinate search, can result in lowering f.

```
basis(i, n) = [k == i ? 1.0 : 0.0 for k in 1 : n]
```

Algorithm 7.1. A function for constructing the `i`th basis vector of length `n`.

Like steepest descent, cyclic coordinate search is guaranteed either to improve or to remain the same with each iteration. No significant improvement after a full cycle over all coordinates indicates that the method has converged. Algorithm 7.2 provides an implementation. As figure 7.2 shows, cyclic coordinate search can fail to find even a local minimum.

```
function cyclic_coordinate_descent(f, x, ϵ)
    Δ, n = Inf, length(x)
    while abs(Δ) > ϵ
        x_orig = x
        for i in 1 : n
            d = basis(i, n)
            x = line_search(f, x, d)
        end
        Δ = norm(x - x_orig)
    end
    return x
end
```

Algorithm 7.2. The cyclic coordinate descent method takes as input the objective function `f` and a starting point `x`, and it runs until the step size over a full cycle is less than a given tolerance `ϵ`.

The method can be augmented with an acceleration step to help traverse diagonal valleys. For every full cycle starting with optimizing $\mathbf{x}^{(1)}$ along $\mathbf{e}^{(1)}$ and ending with $\mathbf{x}^{(n+1)}$ after optimizing along $\mathbf{e}^{(n)}$, an additional line search is conducted along the direction $\mathbf{x}^{(n+1)} - \mathbf{x}^{(1)}$. An implementation is provided in algorithm 7.3 and an example search trajectory is shown in figure 7.3.

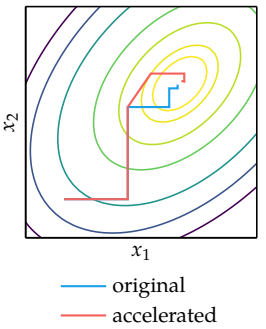


Figure 7.3. Adding the acceleration step to cyclic coordinate descent helps traverse valleys. Six steps are shown for both the original and accelerated versions.

```
function cyclic_coordinate_descent_with_acceleration_step(f, x, ϵ)
    Δ, n = Inf, length(x)
    while abs(Δ) > ϵ
        x_orig = x
        for i in 1 : n
            d = basis(i, n)
            x = line_search(f, x, d)
        end
        x = line_search(f, x, x - x_orig) # acceleration step
        Δ = norm(x - x_orig)
    end
    return x
end
```

Algorithm 7.3. The cyclic coordinate descent method with an acceleration step takes as input the objective function `f` and a starting point `x`, and it runs until the step size over a full cycle is less than a given tolerance `ϵ`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

7.2 Powell's Method

Powell's method[1] can search in directions that are not orthogonal to each other. The method can automatically adjust for long, narrow valleys that might otherwise require a large number of iterations for cyclic coordinate descent or other methods that search in axis-aligned directions.

The algorithm maintains a list of search directions $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$, which are initially the coordinate basis vectors, $\mathbf{u}^{(i)} = \mathbf{e}^{(i)}$ for all i. Starting at $\mathbf{x}^{(1)}$, one iteration of Powell's method conducts a line search for each search direction in succession, updating the design point each time:

$$\mathbf{x}^{(i+1)} \leftarrow \texttt{line\_search}(f, \mathbf{x}^{(i)}, \mathbf{u}^{(i)}) \text{ for } i \text{ in } 1:n \tag{7.3}$$

Next, all search directions are shifted down by one index, dropping the oldest search direction, $\mathbf{u}^{(1)}$:

$$\mathbf{u}^{(i)} \leftarrow \mathbf{u}^{(i+1)} \text{ for } i \text{ in } 1:n-1 \tag{7.4}$$

The last search direction is replaced with the direction from $\mathbf{x}^{(1)}$ to $\mathbf{x}^{(n+1)}$, which is the overall direction of progress over the last cycle:

$$\mathbf{u}^{(n)} \leftarrow \mathbf{x}^{(n+1)} - \mathbf{x}^{(1)} \tag{7.5}$$

and another line search is conducted along the new direction to obtain a new $\mathbf{x}^{(1)}$, completing the iteration. This process is repeated until convergence. Algorithm 7.4 provides an implementation. Figure 7.4 shows an example search trajectory.

Powell showed that for quadratic functions, after k iterations the last k directions will be mutually conjugate. Recall that n line searches along mutually conjugate directions will optimize an n-dimensional quadratic function. Thus, n iterations of Powell's method, totaling $n(n+1)$ line searches, will minimize a quadratic function.

The procedure of dropping the oldest search direction in favor of the overall direction of progress can lead the search directions to become linearly dependent. Without search vectors that are linearly independent, the search directions can no longer cover the full design space, and the method may not be able to find the minimum. This weakness can be mitigated by periodically resetting the search directions to the basis vectors.[2]

[1] Powell's method was first introduced by M. J. D. Powell, "An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives," *Computer Journal*, vol. 7, no. 2, pp. 155–162, 1964. An overview is presented by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1982.

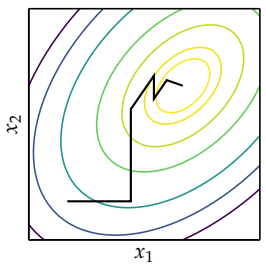


Figure 7.4. Powell's method starts the same as cyclic coordinate descent but iteratively learns conjugate directions.

[2] One recommendation is to reset every n or $n+1$ iterations.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function powell(f, x, ϵ)
    Δ, n = Inf, length(x)
    U = [basis(i,n) for i in 1 : n]
    while Δ > ϵ
        x′ = x
        for i in 1 : n
            d = U[i]
            x′ = line_search(f, x′, d)
        end
        for i in 1 : n-1
            U[i] = U[i+1]
        end
        U[n] = d = x′ - x
        x′ = line_search(f, x′, d)
        Δ = norm(x′ - x)
        x = x′
    end
    return x
end
```

Algorithm 7.4. Powell's method, which takes the objective function `f`, starting point `x`, and tolerance `ϵ`.

7.3 Hooke-Jeeves

The *Hooke-Jeeves method* (algorithm 7.5) traverses the search space based on evaluations at small steps in each coordinate direction.[3] At every iteration, the Hooke-Jeeves method evaluates $f(\mathbf{x})$ and $f(\mathbf{x} \pm \alpha \mathbf{e}^{(i)})$ for a given step size α in every coordinate direction from an *anchoring point* $\mathbf{x}$. It accepts any improvement it may find. If no improvements are found, it will decrease the step size. The process repeats until the step size is sufficiently small. Figure 7.5 shows a few iterations.

[3] R. Hooke and T. A. Jeeves, "Direct Search Solution of Numerical and Statistical Problems," *Journal of the ACM* (*JACM*), vol. 8, no. 2, pp. 212–229, 1961.

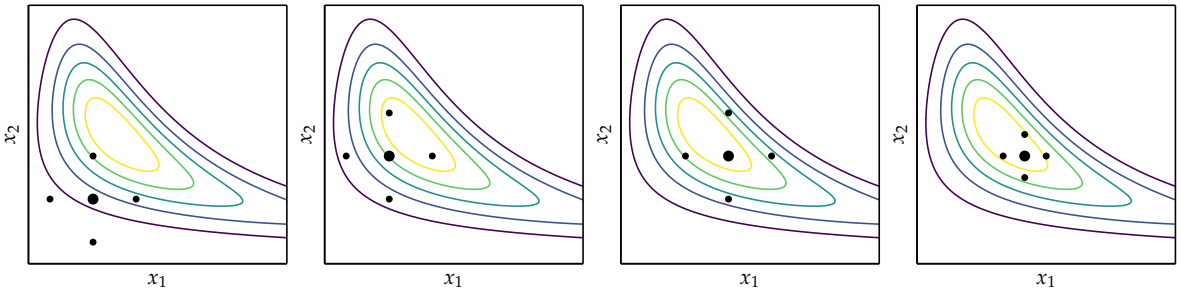


Figure 7.5. The Hooke-Jeeves method, proceeding left to right. It begins with a large step size but then reduces it once it cannot improve by taking a step in any coordinate direction.

One step of the Hooke-Jeeves method requires $2n$ function evaluations for an n-dimensional problem, which can be expensive for problems with many

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

dimensions. The Hooke-Jeeves method is susceptible to local minima. The method has been proven to converge on certain classes of functions.[4]

```
function hooke_jeeves(f, x, α, ϵ, γ=0.5)
    y, n = f(x), length(x)
    while α > ϵ
        improved = false
        best = (x=x,y=y)
        for i in 1 : n
            for sgn in (-1,1)
                x′ = x + sgn*α*basis(i, n)
                y′ = f(x′)
                if y′ < best.y
                    best, improved = (x=x′, y=y′), true
                end
            end
        end
        x, y = best
        if !improved
            α *= γ
        end
    end
    return x
end
```

[4] E. D. Dolan, R. M. Lewis, and V. Torczon, "On the Local Convergence of Pattern Search," *SIAM Journal on Optimization*, vol. 14, no. 2, pp. 567–583, 2003.

Algorithm 7.5. The Hooke-Jeeves method, which takes the target function `f`, a starting point `x`, a starting step size `α`, a tolerance `ϵ`, and a step decay `γ`. When an exploratory search fails to improve the objective, the step size is reduced. The method terminates when the step size is at most `ϵ`. Based on the implementation from A. F. Kaupe Jr., "Algorithm 178: Direct Search," *Communications of the ACM*, vol. 6, no. 6, pp. 313–314, 1963.

7.4 Generalized Pattern Search

In contrast with the Hooke-Jeeves method, which searches in the coordinate directions, *generalized pattern search* can search in arbitrary directions.[5] A *pattern* $\mathcal{P}$ can be constructed from a set of directions $\mathcal{D}$ about an anchoring point $\mathbf{x}$ with a step size α according to:

$$\mathcal{P} = \{\mathbf{x} + \alpha\mathbf{d} \text{ for each } \mathbf{d} \text{ in } \mathcal{D}\} \tag{7.6}$$

The Hooke-Jeeves method uses $2n$ directions for problems in n dimensions, but generalized pattern search can use as few as $n + 1$.

For generalized pattern search to converge to a local minimum, certain conditions must be met. The set of directions must be a *positive spanning set*, which means that we can construct any point in $\mathbb{R}^n$ using a nonnegative linear combination of the directions in $\mathcal{D}$. A positive spanning set ensures that at least one of the directions is a descent direction from a location with a nonzero gradient.[6]

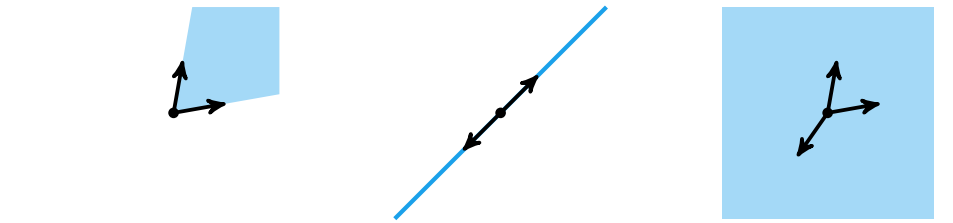
[5] C. Audet and J. E. Dennis Jr., "Mesh Adaptive Direct Search Algorithms for Constrained Optimization," *SIAM Journal on Optimization*, vol. 17, no. 1, pp. 188–217, 2006.

[6] Convergence guarantees for generalized pattern search require that all sampled points fall on a scaled lattice. Each direction must thus be a product $\mathbf{d}^{(j)} = \mathbf{G}\mathbf{z}^{(j)}$ for a fixed nonsingular $n \times n$ matrix $\mathbf{G}$ and integer vector $\mathbf{z}$. V. Torczon, "On the Convergence of Pattern Search Algorithms," *SIAM Journal of Optimization*, vol. 7, no. 1, pp. 1–25, 1997.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can determine whether a given set of directions $\mathcal{D} = \{\mathbf{d}^{(1)}, \mathbf{d}^{(2)}, \ldots, \mathbf{d}^{(m)}\}$ in $\mathbb{R}^n$ is a positive spanning set. First, we construct the matrix $\mathbf{D}$ whose columns are the directions in $\mathcal{D}$ (see figure 7.6). The set of directions $\mathcal{D}$ is a positive spanning set if $\mathbf{D}$ has full row rank and if $\mathbf{Dx} = -\mathbf{D1}$ with $\mathbf{x} \geq \mathbf{0}$ has a solution.[7] This optimization problem is identical to the initialization phase of a linear program, which is covered in chapter 12.

[7] R. G. Regis, "On the Properties of Positive Spanning Sets and Positive Bases," *Optimization and Engineering*, vol. 17, no. 1, pp. 229–262, 2016.



only positively spans the cone

only positively spans 1d space

positively spans $\mathbb{R}^2$

Figure 7.6. A valid pattern for generalized pattern search requires a positive spanning set. These directions are stored in the set $\mathcal{D}$.

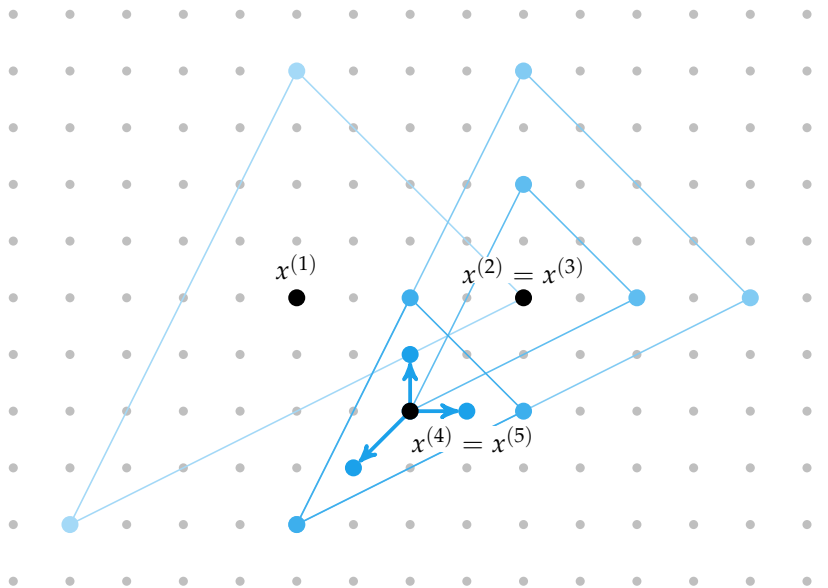


Figure 7.7. All previous points in generalized pattern search lie on a scaled lattice, or mesh. The lattice is not explicitly constructed and need not be axis-aligned.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function generalized_pattern_search(f, x, α, D, ϵ, γ=0.5)
    y = f(x)
    while α > ϵ
        improved = false
        for (i,d) in enumerate(D)
            x′ = x + α*d
            y′ = f(x′)
            if y′ < y
                x, y, improved = x′, y′, true
                D = pushfirst!(deleteat!(D, i), d)
                break
            end
        end
        if !improved
            α *= γ
        end
    end
    return x
end
```

Algorithm 7.6. Generalized pattern search, which takes the target function `f`, a starting point `x`, a starting step size `α`, a set of search directions `D`, a tolerance `ϵ`, and a step decay `γ`. The method reduces the step size whenever none of the points sampled along the search directions provides an improvement, and terminates when the step size is at most `ϵ`.

The implementation of generalized pattern search in algorithm 7.6 contains additional enhancements over the original Hooke-Jeeves method. First, the implementation is *opportunistic*—as soon as an evaluation improves the current best design, it is accepted as the anchoring design point for the next iteration. Second, the implementation uses *dynamic ordering* to accelerate convergence—a direction that leads to an improvement is promoted to the beginning of the list of directions. Figure 7.7 shows a few iterations of the algorithm.

7.5 Nelder-Mead Simplex Method

The *Nelder-Mead simplex method*[8] uses a simplex to traverse the space in search of a minimum. A *simplex* is a generalization of a tetrahedron to n-dimensional space. The Nelder-Mead method uses a series of rules that dictate how the simplex is updated based on evaluations of the objective function at its vertices. Like the Hooke-Jeeves method, the simplex can move around while roughly maintaining its size, and it can shrink as it approaches an optimum.

The simplex consists of the points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n+1)}$. Let $\mathbf{x}_h$ be the vertex with the highest function value, let $\mathbf{x}_s$ be the vertex with the second highest function value, and let $\mathbf{x}_\ell$ be the vertex with the lowest function value. Let $\bar{\mathbf{x}}$ be the mean

[8] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965. We incorporate the improvements in J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 112–147, 1998.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

of all vertices except the highest point $\mathbf{x}_h$. For any design point $\mathbf{x}_\theta$, let $y_\theta = f(\mathbf{x}_\theta)$. A single iteration then evaluates four simplex operations (figure 7.8):

Reflection. $\mathbf{x}_r = \bar{\mathbf{x}} + \alpha(\bar{\mathbf{x}} - \mathbf{x}_h)$, reflects the highest-valued point over the centroid. This typically moves the simplex from high regions toward lower regions. Here, $\alpha > 0$ and is typically set to 1.

Expansion. $\mathbf{x}_e = \bar{\mathbf{x}} + \beta(\mathbf{x}_r - \bar{\mathbf{x}})$, like reflection, but the reflected point is sent even further. This is done when the reflected point has an objective function value less than all points in the simplex. Here, $\beta > \max(1, \alpha)$ and is typically set to 2.

Contraction. $\mathbf{x}_c = \bar{\mathbf{x}} + \gamma(\mathbf{x}_h - \bar{\mathbf{x}})$, the simplex is shrunk down by moving away from the worst point. It is parameterized by $\gamma \in (0, 1)$ which is typically set to 0.5.

Shrinkage. All points are moved toward the best point, typically halving the separation distance.

Figure 7.9 outlines the procedure that is implemented in algorithm 7.7. Figure 7.10 shows several iterations of the algorithm.

The convergence criterion for the Nelder-Mead simplex method is unlike Powell's method in that it considers the variation in the function values rather than the changes to the points in the design space. It compares the standard deviation of the sample $y^{(1)}, \ldots, y^{(n+1)}$ to a tolerance ϵ. This value is high for a simplex over a highly curved region, and it is low for a simplex over a flat region. A highly curved region indicates that there is still further optimization possible.

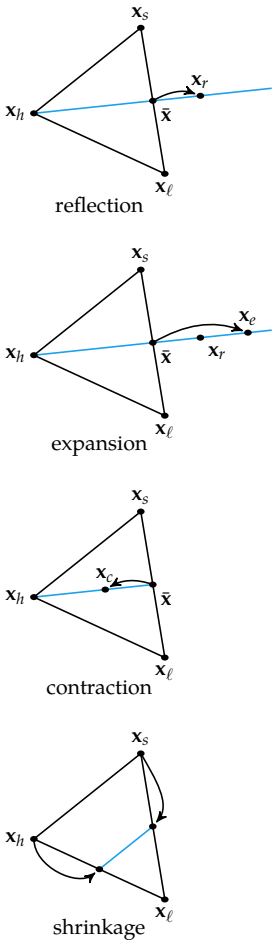


Figure 7.8. The Nelder-Mead simplex operations visualized in two dimensions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

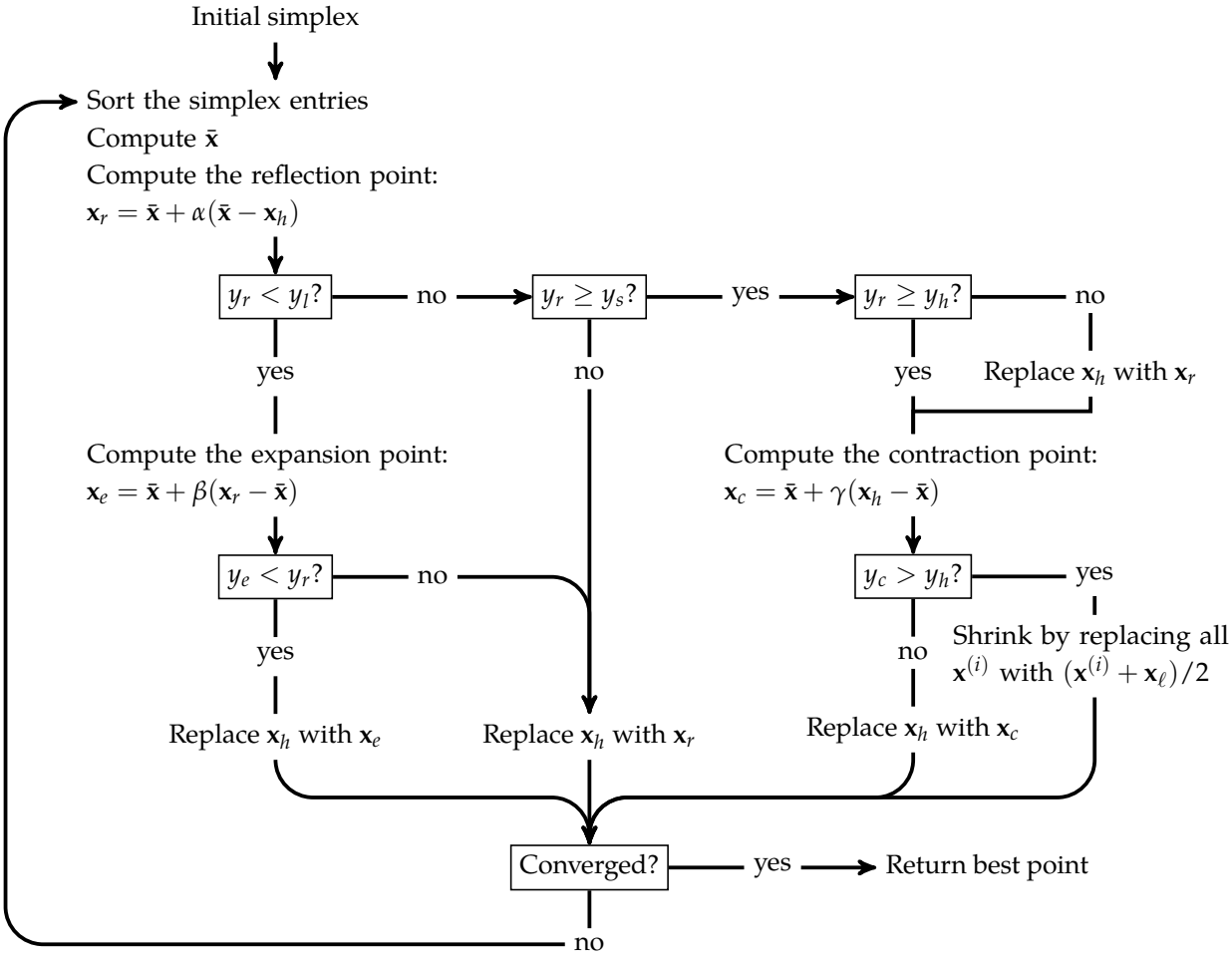


Figure 7.9. Flowchart for the Nelder-Mead algorithm.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

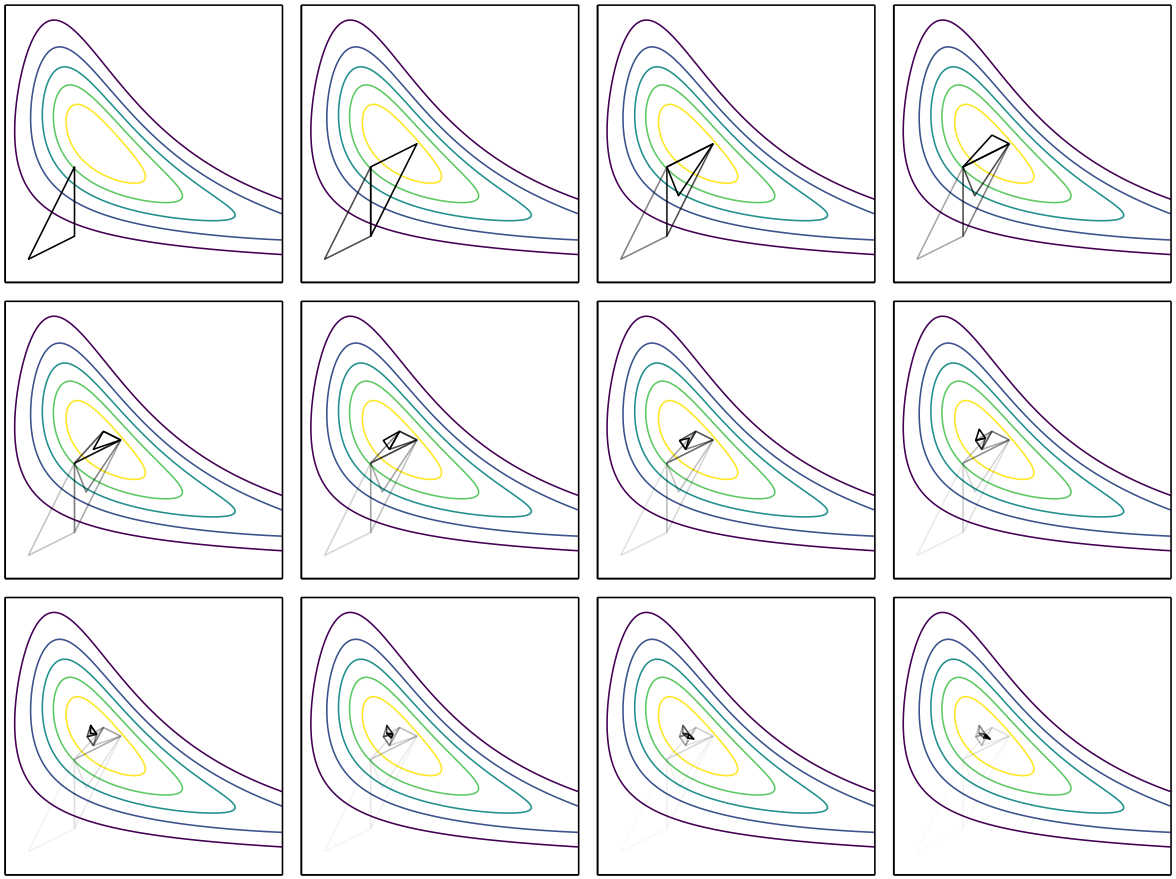


Figure 7.10. The Nelder-Mead method, proceeding left to right and top to bottom.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function nelder_mead(f, S, ϵ; α=1.0, β=2.0, γ=0.5)
    Δ, y_arr = Inf, f.(S)
    while Δ > ϵ
        p = sortperm(y_arr) # sort lowest to highest
        S, y_arr = S[p], y_arr[p]
        xl, yl = S[1], y_arr[1] # lowest
        xh, yh = S[end], y_arr[end] # highest
        xs, ys = S[end-1], y_arr[end-1] # second-highest
        xm = mean(S[1:end-1]) # centroid
        xr = xm + α*(xm - xh) # reflection point
        yr = f(xr)

        if yr < yl
            xe = xm + β*(xr-xm) # expansion point
            ye = f(xe)
            S[end],y_arr[end] = ye < yr ? (xe, ye) : (xr, yr)
        elseif yr ≥ ys
            if yr < yh
                xh, yh, S[end], y_arr[end] = xr, yr, xr, yr
            end
            xc = xm + γ*(xh - xm) # contraction point
            yc = f(xc)
            if yc > yh
                for i in 2 : length(y_arr)
                    S[i] = (S[i] + xl)/2
                    y_arr[i] = f(S[i])
                end
            else
                S[end], y_arr[end] = xc, yc
            end
        else
            S[end], y_arr[end] = xr, yr
        end

        Δ = std(y_arr, corrected=false)
    end
    return S[argmin(y_arr)]
end
```

Algorithm 7.7. The Nelder-Mead simplex method, which takes the objective function `f`, a starting simplex `S` consisting of a list of vectors, and a tolerance `ϵ`. The Nelder-Mead parameters can be specified as well and default to recommended values.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

7.6 Divided Rectangles

The *divided rectangles* algorithm,[9] or *DIRECT* for DIvided RECTangles, incrementally refines a rectangular partition of the design space as illustrated in figure 7.11. The refinement is driven by a heuristic that involves reasoning about potential Lipschitz constants.

To simplify the mathematics and to avoid oversensitivity to dimensions with larger domains, DIRECT first normalizes the search space to be the unit hypercube. If we are minimizing $f(\mathbf{x})$ in the interval between lower and upper ranges $\mathbf{a}$ and $\mathbf{b}$, DIRECT will instead minimize:

$$g(\mathbf{x}) = f(\mathbf{x} \odot (\mathbf{b} - \mathbf{a}) + \mathbf{a}) \tag{7.7}$$

After finding a minimizer $\mathbf{x}^*$ of g, a minimizer of f is

$$\mathbf{x}^* \odot (\mathbf{b} - \mathbf{a}) + \mathbf{a} \tag{7.8}$$

DIRECT maintains a partition of this unit hypercube into hyperrectangular intervals. Each interval has a center $\mathbf{c}^{(i)}$ and an associated objective function value $f(\mathbf{c}^{(i)})$. Each interval also has a radius $r^{(i)}$, which is the distance from the center to a vertex. Figure 7.12 shows such a partition and a plot of the intervals' objective function values at their centers with respect to their radii.

DIRECT begins every iteration by identifying intervals to be split with additional function evaluations. It splits the intervals by reasoning about possible Lipschitz constants. Given a Lipschitz constant ℓ, the lower bound for an interval is a circular cone extending downward from its center $\mathbf{c}^{(i)}$:

$$f(\mathbf{x}) \geq f(\mathbf{c}^{(i)}) - \ell \|\mathbf{x} - \mathbf{c}^{(i)}\|_2 \tag{7.9}$$

This lower bound is constrained by the extents of the interval. Its lowest value is at its vertices, which are all at a distance $r^{(i)}$ from the center with value $f(\mathbf{c}^{(i)}) - \ell r^{(i)}$. Figure 7.13 shows how this minimum value can be seen as the y-intercept of a line of slope ℓ passing through the point $(r^{(i)}, f(\mathbf{c}^{(i)}))$.

Figure 7.14 shows how all such lines can be constructed to find the interval that produces the lowest lower bound for a particular Lipschitz constant. That interval is selected for splitting.

[9] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian Optimization Without the Lipschitz Constant," *Journal of Optimization Theory and Application*, vol. 79, no. 1, pp. 157–181, 1993.

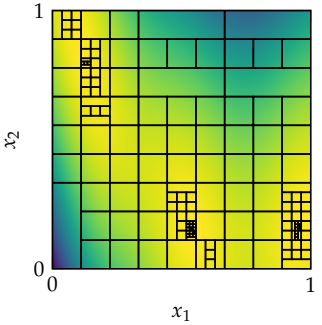


Figure 7.11. The DIRECT method after 16 iterations on the Branin function (appendix B.3). The cells are much denser around the minima of the Branin function because the DIRECT method is designed to increase resolution in promising regions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

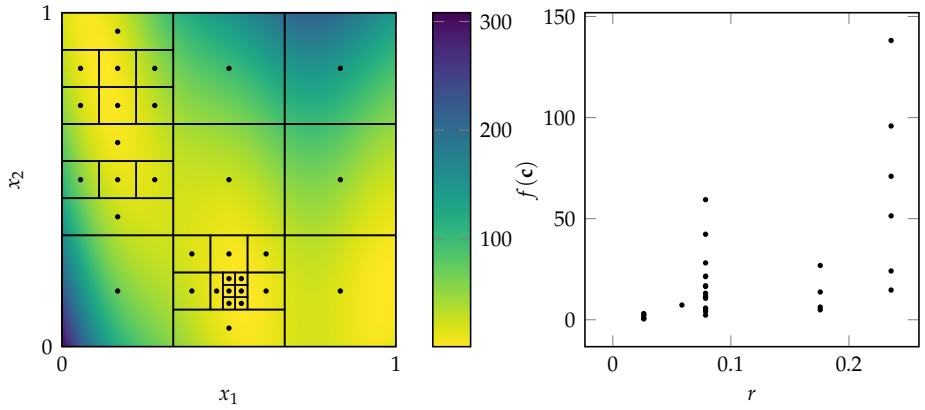


Figure 7.12. The left plot shows the intervals for the DIRECT method after five iterations on the Branin function, appendix B.3. The right plot shows the interval objective function values versus their radii, which is useful for identifying intervals to split with further evaluations.

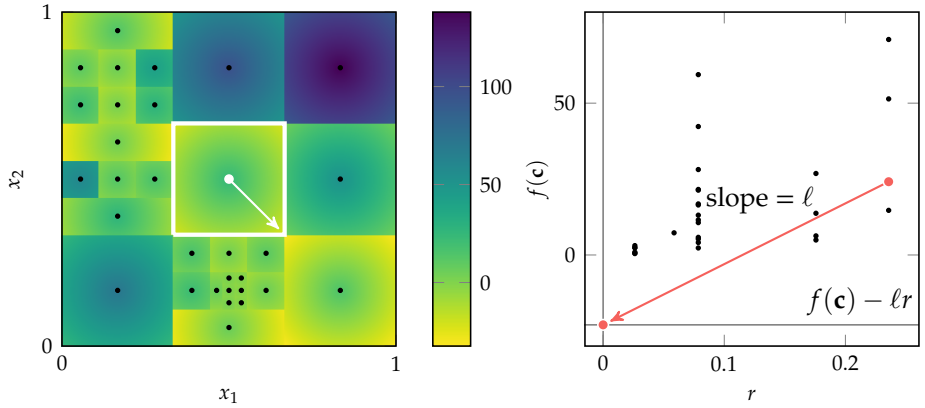


Figure 7.13. The left plot shows the Lipschitz lower bounds constructed for the DIRECT intervals using the Lipschitz constant $\ell = 200$, and highlights one interval. The right plot shows how the minimum value for the lower bound within the highlighted interval is the same as the y-intercept for a line of slope ℓ passing through that interval's $(r, f(\mathbf{c}))$ point.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

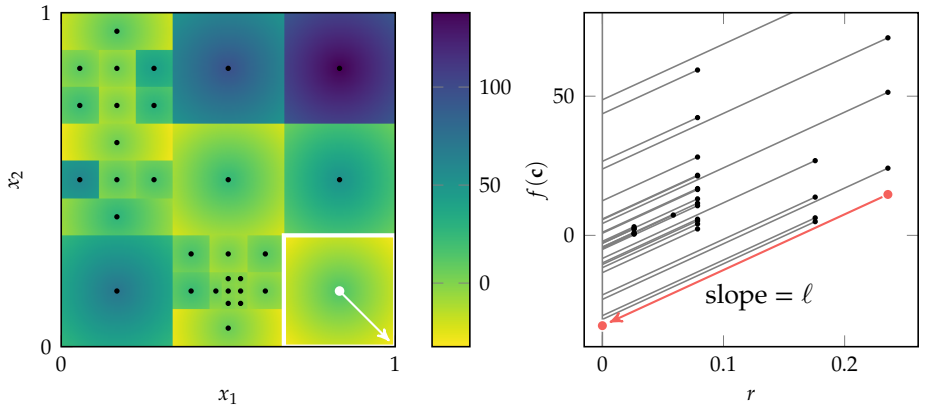


Figure 7.14. The left plot continues to show the Lipschitz lower bounds constructed for the DIRECT intervals using the Lipschitz constant $\ell = 200$, but now highlights the interval containing the lowest value. The right plot shows how the lowest lower bound for a given Lipschitz constant is the one with the lowest y-intercept in the right-hand plot.

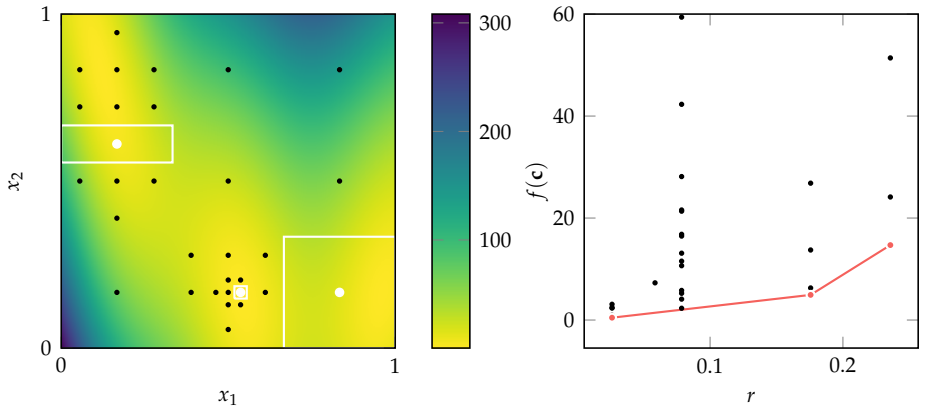


Figure 7.15. The left plot shows the split intervals identified for this iteration of DIRECT on the Branin function. The right plot shows the lower-right convex hull formed by the points associated with these intervals in $(r, f(\mathbf{c}))$ space.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The DIRECT method does not operate on just one value for the Lipschitz constant, but selects all intervals for which a Lipschitz constant exists such that their lower bounds have minimal value. These split intervals form a piecewise-linear boundary[10] along the lower-right of the $(r, f(\mathbf{c}))$ space, as shown in figure 7.15.

[10] These points are the lower-right convex hull.

The selected intervals are split into thirds along the axis directions. The order in which we split an interval's dimensions matters, as shown in figure 7.16. DIRECT will choose a split order such that lower (better) function evaluations receive larger sub-rectangles, encouraging their selection for later splitting. When splitting a region without equal side lengths, only the longest dimensions are split (figure 7.17).

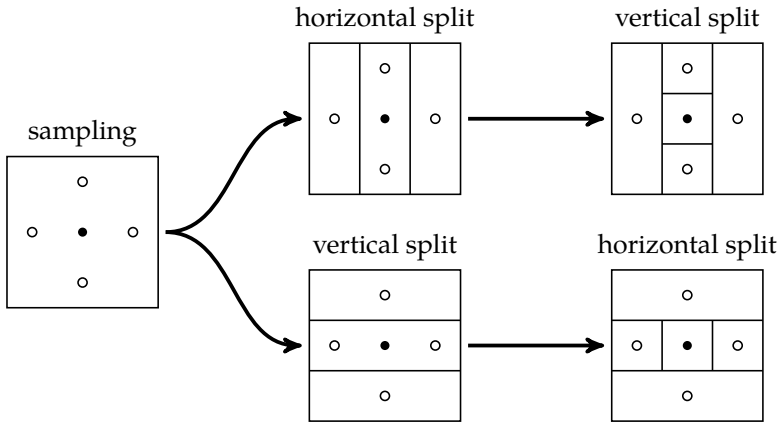


Figure 7.16. Interval splitting in multiple dimensions for DIRECT requires choosing an ordering for the split dimensions.

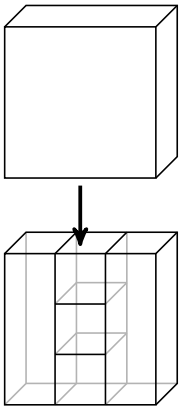


Figure 7.17. DIRECT will only split the longest dimensions of intervals.

The width in a given dimension depends on how many times that dimension has been split. Since DIRECT always splits axis directions by thirds, a dimension that has been split d times will have a width of 3^{-d}. If we have n dimensions and track how many times each dimension of a given interval has been split in a vector $\mathbf{d}$, then the radius of that interval is

$$r = \left\| \left[\frac{1}{2 \cdot 3^{d_1}}, \cdots, \frac{1}{2 \cdot 3^{d_n}} \right] \right\|_2 \tag{7.10}$$

DIRECT is implemented in algorithm 7.8. Iterations of DIRECT begin by finding all split intervals (algorithm 7.9).[11] It then splits all split intervals (algorithm 7.10). Two iterations of DIRECT in two dimensions are demonstrated in example 7.1.

[11] The algorithm only divides intervals larger than a minimum radius. This minimum radius prevents inefficient function evaluations very close to existing points.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct DirectRectangle
    c # center point
    y # center point value
    d # number of divisions per dimension
    r # the radius of the interval
end

function direct(f, a, b, k_max, r_min)
    g = x -> f(x.*(b-a) + a) # evaluate within unit hypercube

    n = length(a)
    c = fill(0.5, n)
    □s = [DirectRectangle(c, g(c), fill(0, n), sqrt(n)/2)]

    c_best = c
    for k in 1 : k_max
        □s_split = direct_split_intervals(□s, r_min)
        setdiff!(□s, □s_split)
        for □_split in □s_split
            append!(□s, split_interval(□_split, g))
        end
        c_best = □s[findmin(□.y for □ in □s)[2]].c
    end

    return c_best.*(b-a) + a # from unit hypercube
end
```

Algorithm 7.8. DIRECT, which takes the multidimensional objective function `f`, vector of lower bounds `a`, vector of upper bounds `b`, number of iterations `k_max`, and minimum interval radius `r_min`. It returns the best coordinate. DIRECT maintains a set of hyperrectangular intervals defined by the `DirectRectangle` structure.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function is_ccw(a, b, c) # is a→b→c counter-clockwise
    return a.r*(b.y-c.y)-a.y*(b.r-c.r)+(b.r*c.y-b.y*c.r) < 1e-6
end

function direct_split_intervals(□s, r_min)
    hull = DirectRectangle[]
    # Sort the rects by increasing r, then by increasing y
    sort!(□s, by = □ -> (□.r, □.y))
    for □ in □s
        if length(hull) ≥ 1 && □.r == hull[end].r
            # Repeated r values cannot be improvements
            continue
        end
        if length(hull) ≥ 1 && □.y ≤ hull[end].y
            # Remove the last point if the new one is better
            pop!(hull)
        end
        if length(hull) ≥ 2 && is_ccw(hull[end-1], hull[end], □)
            # Remove the last point if the new one is better
            pop!(hull)
        end
        push!(hull, □)
    end
    # Only split intervals larger than the minimum radius
    filter!(□ -> □.r ≥ r_min, hull)
    return hull
end
```

Algorithm 7.9. A routine for obtaining the split intervals from a given list of `DirectRectangle`s `□s` and a minimum radius `r_min`. The potentially optimal intervals form a lower-right convex hull in radius r and objective y.

```
function split_interval(□, g)
    c, n, d_min, d = □.c, length(□.c), minimum(□.d), copy(□.d)
    dirs, δ = findall(d .== d_min), 3.0^(-d_min-1)
    # Sample the objective function in all split directions,
    # and track the minimum value in each axis.
    Cs = [(c + δ*basis(i,n), c - δ*basis(i,n)) for i in dirs]
    Ys = [(g(C[1]), g(C[2])) for C in Cs]
    minvals = [min(Y[1], Y[2]) for Y in Ys]

    # Split the axes in order by increasing minimum value.
    □s = DirectRectangle[]
    for j in sortperm(minvals)
        d[dirs[j]] += 1 # increment the number of splits
        C, Y, r = Cs[j], Ys[j], norm(0.5*3.0.^(-d))
        push!(□s, DirectRectangle(C[1], Y[1], copy(d), r))
        push!(□s, DirectRectangle(C[2], Y[2], copy(d), r))
    end
    r = norm(0.5*3.0.^(-d))
    push!(□s, DirectRectangle(c, □.y, d, r))
    return □s
end
```

Algorithm 7.10. The routine for splitting an interval `□`, where `g` is the objective function in the unit hypercube. It returns a list of the resulting smaller intervals.

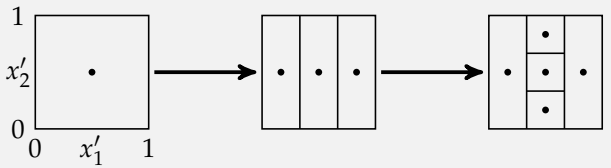
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 7.1. The first two iterations of DIRECT on the flower function (appendix B.4).

We can use DIRECT to optimize the flower function over $x_1 \in [0,3]$, $x_2 \in [-2,1]$. The function is first normalized to the unit hypercube such that $x'_1, x'_2 \in [0,1]$:

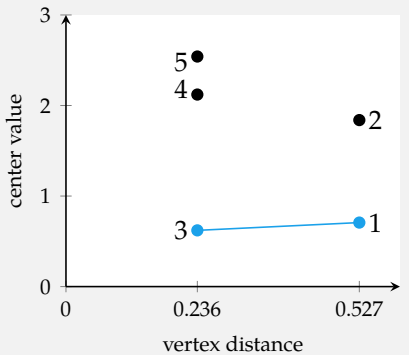
$$f(x'_1, x'_2) = \text{flower}(3x'_1, 3x'_2 - 2)$$

The objective function is sampled at $[0.5, 0.5]$ to obtain 0.621. We have a single interval with center $[0.5, 0.5]$ and side lengths $[1, 1]$. The interval is divided twice, first into thirds in x'_1 and then the center interval is divided into thirds in x'_2.



We now have five intervals:

interval	center	side lengths	vertex distance	center value
1	$[1/6, 3/6]$	$[1/3, 1]$	0.527	0.707
2	$[5/6, 3/6]$	$[1/3, 1]$	0.527	1.839
3	$[3/6, 3/6]$	$[1/3, 1/3]$	0.236	0.621
4	$[3/6, 1/6]$	$[1/3, 1/3]$	0.236	2.121
5	$[3/6, 5/6]$	$[1/3, 1/3]$	0.236	2.541



We next split on intervals 1 and 3.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

7.7 Summary

- Direct methods rely solely on the objective function and do not use derivative information.
- Cyclic coordinate search optimizes one coordinate direction at a time.
- Powell's method adapts search directions based on the direction of progress.
- Hooke-Jeeves searches in each coordinate direction from the current point using a step size that is adapted over time.
- Generalized pattern search is similar to Hooke-Jeeves, but it uses fewer search directions that positively span the design space.
- The Nelder-Mead simplex method uses a simplex to search the design space, adaptively expanding and contracting the size of the simplex in response to evaluations of the objective function.
- The divided rectangles algorithm uses a heuristic inspired by potential Lipschitz constants to iteratively refine a rectangular partition of the design space.

7.8 Exercises

Exercise 7.1. Previous chapters covered methods that leverage the derivative to descend toward a minimum. Direct methods are able to use only zero-order information—evaluations of f. How many evaluations are needed to approximate the derivative and the Hessian of an n-dimensional objective function using finite difference methods? Why do you think it is important to have zero-order methods?

Solution: The derivative has n terms whereas the Hessian has n^2 terms. Each derivative term requires two evaluations when using finite difference methods: $f(\mathbf{x})$ and $f(\mathbf{x} + h\mathbf{e}^{(i)})$. Each Hessian term requires an additional evaluation when using finite difference methods:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \approx \frac{f(\mathbf{x} + h\mathbf{e}^{(i)} + h\mathbf{e}^{(j)}) - f(\mathbf{x} + h\mathbf{e}^{(i)}) - f(\mathbf{x} + h\mathbf{e}^{(j)}) + f(x)}{h^2}$$

Thus, to approximate the gradient, you need $n + 1$ evaluations, and to approximate the Hessian you need on the order of n^2 evaluations.

Approximating the Hessian is prohibitively expensive for large n. Direct methods can take comparatively more steps using n^2 function evaluations, as direct methods need not estimate the derivative or Hessian at each step.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 7.2. Design an objective function and a starting point such that Hooke-Jeeves will fail to reduce the objective function.

Solution: Consider minimizing $f(\mathbf{x}) = x_1 x_2$ from the starting point $[0, 0]$. Proceeding in either canonical direction will not reduce the objective function, but $[0, 0]$ is clearly not a minimizer.

Exercise 7.3. Is the design point obtained using the Hooke-Jeeves method guaranteed to be within ϵ of a local minimum?

Solution: At each iteration, the Hooke-Jeeves method samples $2n$ points along the coordinate directions with a step-size α. It stops when none of the points provides an improvement and the step size is no more than a given tolerance ϵ. While this often causes the Hooke-Jeeves method to stop when it has converged to within ϵ of a local minimum, that need not be the case. For example, a valley can descend between two coordinate directions farther than ϵ before arriving at a local minimum, and the Hooke-Jeeves method would not detect it.

Exercise 7.4. Give an example of a concrete engineering problem where you may not be able to compute analytical derivatives.

Solution: Minimizing the drag of an airfoil subject to a minimum thickness (to preserve structural integrity). Evaluating the performance of the airfoil using computational fluid dynamics involves solving partial differential equations. Because the function is not known analytically, we are unlikely to have an analytical expression for the derivative.

Exercise 7.5. State a difference between the divided rectangles algorithm in one dimension and the Shubert-Piyavskii method.

Solution: The divided rectangles method samples at the center of the intervals and not where the bound derived from a known Lipschitz constant is lowest.

Exercise 7.6. Suppose our search algorithm has us transition from $\mathbf{x}^{(k)} = [1, 2, 3, 4]$ to $\mathbf{x}^{(k+1)} = [2, 2, 2, 4]$. Could our search algorithm be (a) cyclic coordinate search, (b) Powell's method, (c) both a and b, or (d) neither a nor b? Why?

Solution: It cannot be cyclic coordinate search since more than one component is changing. It can be Powell's method.

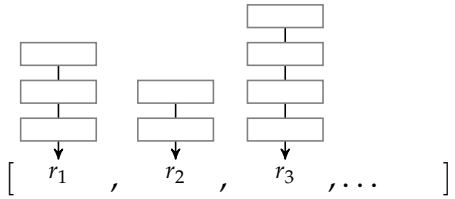
Exercise 7.7. The current implementation of `direct_split_intervals` in algorithm 7.9 iterates over all intervals. Knowing that many intervals will share the same radius, is there a data structure that could be used to make finding the splitting intervals more efficient?

Solution: Finding the split intervals requires finding those with points $(f(\mathbf{c}), r)$ furthest to the bottom-right. If two intervals have the same radius r, then one with a higher objective function value cannot possibly lie in the bottom-right. Rather than storing the intervals in an array:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



the recommended implementation for DIRECT maintains a separate y-based priority queue for each distinct value of r:



The priority queues are sorted in increasing order by radius. This approach allows for efficiently accessing and keeping track of the lowest interval for each radius.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

8 Stochastic Methods

This chapter presents a variety of *stochastic methods* that use randomization strategically to help explore the design space for an optimum. Randomness can help escape local optima and increase the chances of finding a global optimum. Stochastic methods typically use *pseudo-random* number generators to ensure repeatability.[1] Pure random search is often inefficient because it does not exploit information from previous evaluations. This chapter discusses a variety of ways to control the degree of randomness in our search.

[1] Although pseudo-random number generators produce numbers that appear random, they are actually a result of a deterministic process. Pseudo-random numbers can be produced through calls to the `rand` function. The process can be reset to an initial state using the `seed!` function from the `Random.jl` package.

8.1 Noisy Descent

Adding stochasticity to gradient descent can be beneficial in large nonlinear optimization problems. Saddle points, where the gradient is very close to zero, can cause descent methods to select step sizes that are too small to be useful. One approach is to add Gaussian noise to each descent step $\mathbf{d}^{(k)}$:[2]

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \mathbf{d}^{(k)} + \boldsymbol{\epsilon}^{(k)} \tag{8.1}$$

where $\boldsymbol{\epsilon}^{(k)}$ is zero-mean Gaussian noise with standard deviation σ. The amount of noise is typically reduced over time. The standard deviation of the noise is typically a decreasing sequence $\sigma^{(k)}$ such as $1/k$.[3] Algorithm 8.1 provides an implementation of this method. Figure 8.1 compares descent with and without noise on a saddle function.

[2] G. Hinton and S. Roweis, "Stochastic Neighbor Embedding," in *Advances in Neural Information Processing Systems* (*NIPS*), 2003.

[3] The Hinton and Roweis paper used a fixed standard deviation for the first 3,500 iterations and set the standard deviation to zero thereafter.

A common approach for training neural networks is *stochastic gradient descent*, which uses a noisy gradient approximation. In addition to helping traverse past saddle points, evaluating noisy gradients using randomly chosen subsets of the training data[4] is significantly less expensive computationally than calculating the true gradient at every iteration.

[4] These subsets are called *batches*.

```
mutable struct NoisyDescent <: DescentMethod
    submethod # descent method to apply noise to
    σ         # noise sequence
    k         # iteration
end
function init!(M::NoisyDescent, f, ∇f, x)
    init!(M.submethod, f, ∇f, x)
    M.k = 1
    return M
end
function step!(M::NoisyDescent, f, ∇f, x)
    x = step!(M.submethod, f, ∇f, x)
    σ = M.σ(M.k)
    x += σ.*randn(length(x))
    M.k += 1
    return x
end
```

Algorithm 8.1. A noisy descent method, which augments another descent method with additive Gaussian noise. The method takes another `DescentMethod` `submethod`, a noise sequence `σ`, and stores the iteration count `k`.

Convergence guarantees for stochastic gradient descent require that the positive step sizes be chosen such that:

$$\sum_{k=1}^{\infty} \alpha^{(k)} = \infty \qquad \sum_{k=1}^{\infty} \left(\alpha^{(k)}\right)^2 < \infty \tag{8.2}$$

These conditions ensure that the step sizes decrease and allow the method to converge, but not too quickly so as to become stuck away from a local minimum.[5]

[5] The effect of step factor decay on convergence was covered in section 4.2.

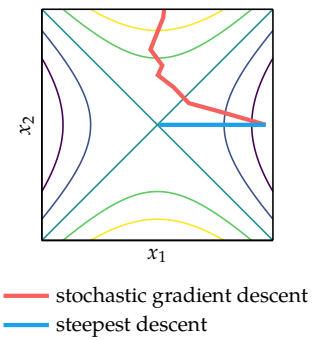


Figure 8.1. Adding stochasticity to a descent method helps with traversing saddle points such as $f(\mathbf{x}) = x_1^2 - x_2^2$ shown here. Due to the initialization, the steepest descent method converges to the saddle point where the gradient is zero.

8.2 Mesh Adaptive Direct Search

The generalized pattern search methods covered in section 7.4 restricted local exploration to a fixed set of directions. In contrast, *mesh adaptive direct search* uses random positive spanning directions.[6] The mesh referred to in the name of this method consists of the points in the design space that are reachable by taking steps in these spanning directions.

The procedure used to sample positive spanning sets (see example 8.1) begins by constructing an initial linearly spanning set in the form of a lower triangular matrix $\mathbf{L}$. The diagonal terms in $\mathbf{L}$ are sampled from $\pm 1/\sqrt{\alpha^{(k)}}$, where $\alpha^{(k)}$ is the step size at iteration k. The lower components of $\mathbf{L}$ are sampled from

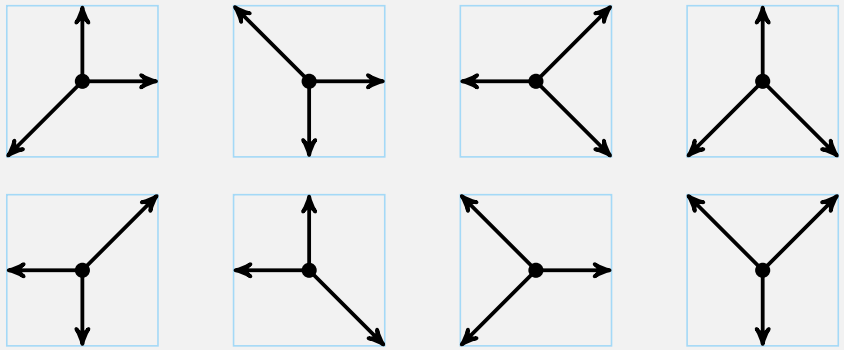
$$\left\{-1/\sqrt{\alpha^{(k)}} + 1, -1/\sqrt{\alpha^{(k)}} + 2, \ldots, 1/\sqrt{\alpha^{(k)}} - 1\right\} \tag{8.3}$$

[6] This section follows the lower triangular mesh adaptive direct search given by C. Audet and J. E. Dennis Jr., "Mesh Adaptive Direct Search Algorithms for Constrained Optimization," *SIAM Journal on Optimization*, vol. 17, no. 1, pp. 188–217, 2006.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 8.1. Positive spanning sets for $\mathbb{R}^2$. Note that the lower triangular generation strategy can only generate the first two columns of spanning sets.

Consider positive spanning sets constructed from the nonzero directions $d_1, d_2 \in \{-1, 0, 1\}$. There are 8 positive spanning sets with 3 elements that can be constructed from these directions:



The rows and columns of $\mathbf{L}$ are then randomly permuted to obtain a matrix $\mathbf{D}$ whose columns correspond to n directions that linearly span $\mathbb{R}^n$. Each of these directions is inscribed inside a hypercube of side length $2/\sqrt{\alpha^{(k)}}$.[7]

[7] For each direction, $\|\mathbf{d}^{(i)}\|_\infty = 1/\sqrt{\alpha^{(k)}}$.

Two common methods for obtaining a positive spanning set from the linearly spanning set are to add one additional direction $\mathbf{d}^{(n+1)} = -\sum_{i=1}^{n} \mathbf{d}^{(i)}$ and to add n additional directions $\mathbf{d}^{(n+j)} = -\mathbf{d}^{(j)}$ for j in $1:n$. We use the first method in algorithm 8.2.

The step size α starts at 1, is always a power of 4, and never exceeds 1. Using a power of 4 causes the maximum possible componentwise step size taken in each iteration to be scaled by a factor of 2, as the maximum componentwise step size $\alpha/\sqrt{\alpha}$ has length $4^m/\sqrt{4^m} = 2^m$ for integer $m < 1$. The step size is updated according to:

$$\alpha^{(k+1)} \leftarrow \begin{cases} \alpha^{(k)}/4 & \text{if no improvement was found in this iteration} \\ \min(1, 4\alpha^{(k)}) & \text{otherwise} \end{cases} \tag{8.4}$$

Mesh adaptive direct search is opportunistic and does not support dynamic ordering[8] because, after a successful iteration, the new random pattern may not contain the previous successful direction. The algorithm queries a new design point along the accepted descent direction. If $f(\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \alpha\mathbf{d}) < f(\mathbf{x}^{(k-1)})$,

[8] See section 7.4.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function rand_positive_spanning_set(α, n)
    δ = round(Int, 1/sqrt(α))
    L = Matrix(Diagonal(δ*rand([1,-1], n)))
    for i in 2:n
        for j in 1:i-1
            L[i,j] = rand(-δ+1:δ-1)
        end
    end
    D = L[randperm(n),:]
    D = D[:,randperm(n)]
    D = [D  -sum(D,dims=2)]
    return [D[:,i] for i in 1 : n+1]
end
```

Algorithm 8.2. Randomly sampling a positive spanning set of $n+1$ directions according to mesh adaptive direct search with step size `α` and number of dimensions `n`.

then the queried point is $\mathbf{x}^{(k-1)} + 4\alpha\mathbf{d} = \mathbf{x}^{(k)} + 3\alpha\mathbf{d}$. The procedure is outlined in algorithm 8.3. Figure 8.2 illustrates how this algorithm explores the search space.

```
function mesh_adaptive_direct_search(f, x, ϵ)
    α, y, n = 1, f(x), length(x)
    while α > ϵ
        improved = false
        for (i,d) in enumerate(rand_positive_spanning_set(α, n))
            x′ = x + α*d
            y′ = f(x′)
            if y′ < y
                x, y, improved = x′, y′, true
                x′ = x + 3α*d
                y′ = f(x′)
                if y′ < y
                    x, y = x′, y′
                end
                break
            end
        end
        α = improved ? min(4α, 1) : α/4
    end
    return x
end
```

Algorithm 8.3. Mesh adaptive direct search for an objective function f, an initial design $\mathbf{x}$, and a tolerance `ϵ`.

8.3 *Memory-Efficient Zeroth-Order Optimization*

Memory efficiency becomes a concern when optimizing very large designs, such as deep learning models with many billions of parameters that push the bound-

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

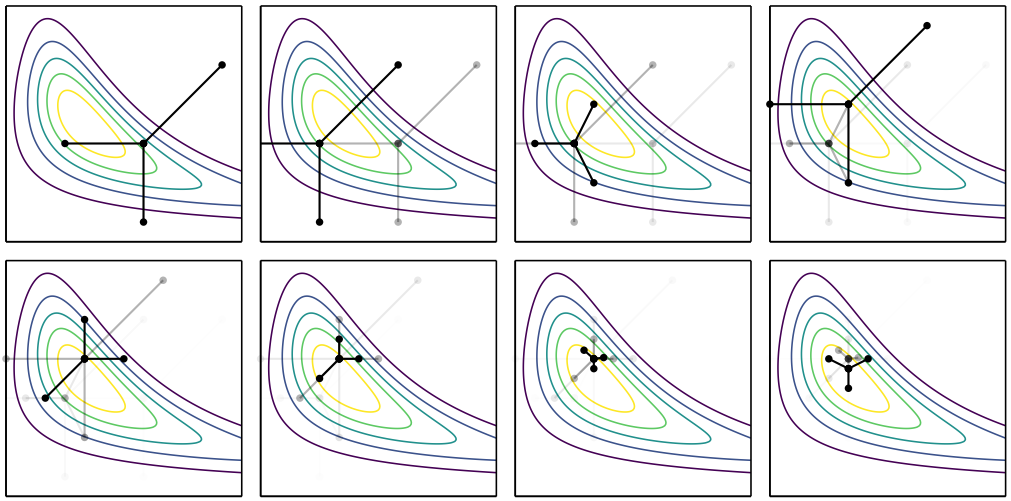


Figure 8.2. Mesh adaptive direct search proceeding left to right and top to bottom.

aries of memory storage.[9] Stochastic methods are not merely practical, but can be helpful because the exact computation of a gradient may be prohibitively expensive or infeasible. The *memory-efficient zeroth-order optimizer* (*MeZO*) computes a stochastic gradient step that can be estimated and applied in-place, using the same memory footprint used to store the design vector. This allows for the optimization of far larger problems.[10]

This approach uses the simultaneous perturbation stochastic gradient approximation (section 2.6), which approximates the gradient using directional derivatives along randomly chosen directions.[11] In MeZO, the gradient is estimated and a step is applied in-place such that a separate gradient vector does not need to be allocated, thereby minimizing the memory footprint. The update is as follows:

$$\mathbf{x}' = \mathbf{x} - \alpha \frac{f(\mathbf{x} + \delta\mathbf{z}) - f(\mathbf{x} - \delta\mathbf{z})}{2\delta}\mathbf{z} \tag{8.5}$$

for a step factor α, finite difference scalar δ, and perturbation $\mathbf{z}$ drawn from a zero-mean unit Gaussian distribution. Such an update is called a *zero-order stochastic step*.

The zero-order stochastic step is implemented in algorithm 8.4. We start by setting our pseudo-random number generator according to the seed, and then randomly perturb our design such that it contains $\mathbf{x} + \delta\mathbf{z}$. We then evaluate the objective function to get $f(\mathbf{x} + \delta\mathbf{z})$. We again reset the pseudo-random number

[9] For example, the Llama 3.1 large language model has 405 billion parameters. A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, et al., "The Llama 3 Herd of Models," 2024. arXiv: 2407.21783.

[10] In the context of training large language models, this algorithm has been shown to lead to a 12× memory reduction and a 2× speed up in training time. S. Malladi, T. Gao, E. Nichani, A. Damian, J. Lee, D. Chen, et al., "Fine-Tuning Language Models with Just Forward Passes," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2023.

[11] The simultaneous perturbation stochastic gradient approximation can average over multiple sampled perturbations. It is common to use only a single sample perturbation per step for MeZO.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

generator to the same seed value, and again perturb the design, but this time by twice as much in the opposite direction. Our design will thereafter contain $\mathbf{x} - \delta\mathbf{z}$, and we can evaluate $f(\mathbf{x} - \delta\mathbf{z})$. We return our design to its initial value by again resetting to the seed and again perturbing by $\delta\mathbf{z}$. The step is applied using another perturbation of the appropriate magnitude.

```
function perturb_parameters!(x, δ, seed)
    Random.seed!(seed)
    for i in eachindex(x)
        x[i] += δ*randn()
    end
end

function zero_order_stochastic_step!(x, f, δ, α, seed)
    # positive perturbation
    perturb_parameters!(x, δ, seed)
    y = f(x)

    # negative perturbation
    perturb_parameters!(x, -2δ, seed)
    Δy = y - f(x)

    # recover the original parameters
    perturb_parameters!(x, δ, seed)

    # apply a gradient step
    perturb_parameters!(x, -α * Δy / (2δ), seed)

    return x
end
```

Algorithm 8.4. A method for improving a design `x` in-place based on a single simultaneous perturbation stochastic gradient estimate, without allocating additional memory. Perturbations are zero-mean with standard deviation `δ` using the provided seed. The method directly applies a step with step size `α` to the parameters. Successive calls to this method should provide different random seeds.

8.4 Simulated Annealing

Simulated annealing[12] borrows inspiration from metallurgy.[13] *Temperature* is used to control the degree of stochasticity during the randomized search. The temperature starts high, allowing the process to freely move about the search space, with the hope that in this phase the process will find a good region with the best local minimum. The temperature is then slowly brought down, reducing the stochasticity and forcing the search to converge to a minimum. Simulated annealing is often used on functions with many local minima due to its ability to escape local minima.

[12] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[13] Annealing is a process in which a material is heated and then cooled, making it more workable. When hot, the atoms in the material are more free to move around, and, through random motion, tend to settle into better positions. A slow cooling brings the material to an ordered, crystalline state. A fast, abrupt quenching causes defects because the material is forced to settle in its current condition.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

At every iteration, a candidate transition from $\mathbf{x}$ to $\mathbf{x}'$ is sampled from a transition distribution T and is accepted with probability

$$\begin{cases} 1 & \text{if } \Delta y \leq 0 \\ e^{-\Delta y/t} & \text{if } \Delta y > 0 \end{cases} \tag{8.6}$$

where $\Delta y = f(\mathbf{x}') - f(\mathbf{x})$ is the difference in the objective and $t > 0$ is the temperature. This acceptance probability, known as the *Metropolis criterion*,[14] allows the algorithm to escape from local minima when the temperature is high.

The temperature parameter t controls the acceptance probability. An annealing schedule is used to slowly bring down the temperature as the algorithm progresses, as illustrated by figure 8.3. The temperature must be brought down to ensure convergence. If it is brought down too quickly, the search method may not cover the portion of the search space containing the global minimum.

It can be shown that a *logarithmic annealing schedule* of $t^{(k)} = t^{(1)} \ln(2) / \ln(k+1)$ for the kth iteration is guaranteed to asymptotically reach the global optimum under certain conditions,[15] but it can be very slow in practice. The *exponential annealing schedule*, which is more common, uses a simple decay factor $t^{(k+1)} = \gamma t^{(k)}$ for some $\gamma \in (0, 1)$. Another common annealing schedule, *fast annealing*,[16] uses a temperature of $t^{(k)} = \frac{t^{(1)}}{k}$. A basic implementation of simulated annealing is provided by algorithm 8.5. Example 8.2 shows the effect different transition distributions and annealing schedules have on the optimization process.

```
function simulated_annealing(f, x, T, t, k_max)
    y = f(x)
    best = (x=x, y=y)
    for k in 1 : k_max
        x′ = x + rand(T)
        y′ = f(x′)
        Δy = y′ - y
        if Δy ≤ 0 || rand() < exp(-Δy/t(k))
            x, y = x′, y′
        end
        if y′ < best.y
            best = (x=x′, y=y′)
        end
    end
    return best.x
end
```

[14] Named for the Greek-American physicist Nicholas Metropolis (1915–1999).

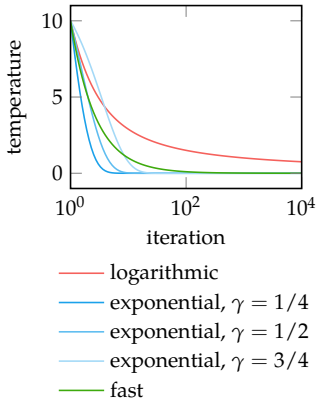


Figure 8.3. Several annealing schedules commonly used in simulated annealing. The schedules have an initial temperature of 10.

[15] B. Hajek, "Cooling Schedules for Optimal Annealing," *Mathematics of Operations Research*, vol. 13, no. 2, pp. 311–329, 1988.

[16] H. Szu and R. Hartley, "Fast Simulated Annealing," *Physics Letters A*, vol. 122, no. 3–4, pp. 157–162, 1987.

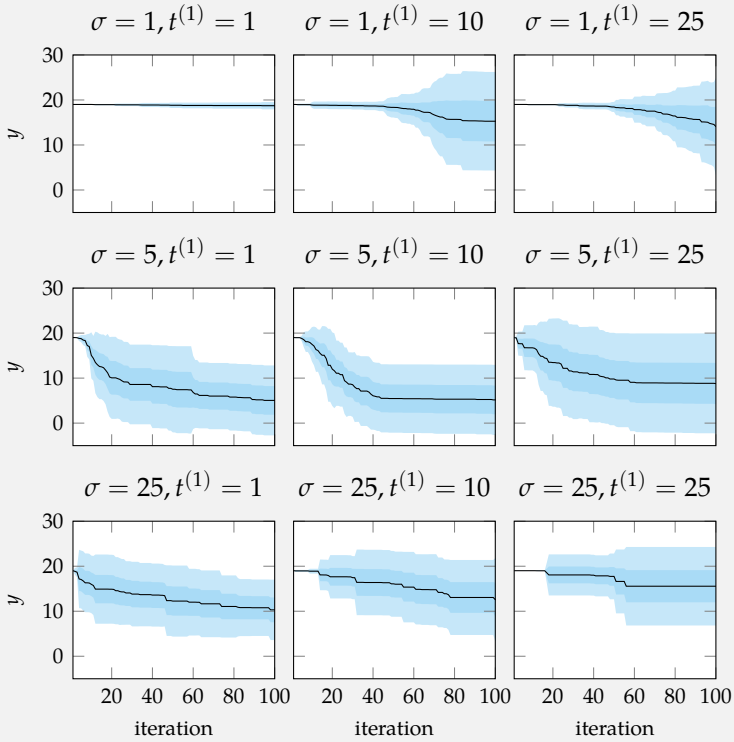
Algorithm 8.5. Simulated annealing, which takes as input an objective function `f`, an initial point `x`, a transition distribution `T`, an annealing schedule `t`, and the number of iterations `k_max`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 8.2. Exploring the effect of distribution variance and temperature on the performance of simulated annealing. The blue regions indicate the 5% to 95% and 25% to 75% empirical Gaussian quantiles of the objective function value.

We can use simulated annealing to optimize Ackley's function (appendix B.1). Ackley's function has many local minima, making it easy for gradient-based methods to get stuck.

Suppose we start at $\mathbf{x}^{(1)} = [15, 15]$ and run 100 iterations. Below we show the distribution over iterations for multiple runs with different combinations of three zero-mean, diagonal covariance ($\sigma\mathbf{I}$) Gaussian transition distributions, and three different temperature schedules $t^{(k)} = t^{(1)}/k$.



In this case, the spread of the transition distribution has the greatest impact on performance.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A more sophisticated algorithm was introduced by Corana, Marchesi, Martini, and Ridella in 1987 that allows for the step size to change during the search.[17] Rather than using a fixed transition distribution, this adaptive simulated annealing method keeps track of a separate step size $\mathbf{v}$ for each coordinate direction. For a given point $\mathbf{x}$, a cycle of random moves is performed in each coordinate direction i according to:

$$\mathbf{x}' = \mathbf{x} + r v_i \mathbf{e}_i \tag{8.7}$$

where r is drawn uniformly at random from $[-1, 1]$ and v_i is the maximum step size in the ith coordinate direction. Each new point is accepted according to the Metropolis criterion. The number of accepted points in each coordinate direction is stored in a vector $\mathbf{a}$.

After n_s cycles, the step sizes are adjusted with the aim to maintain an approximately equal number of accepted and rejected designs with an average acceptance rate near one-half. Rejecting too many moves is a waste of computational effort, while accepting too many moves indicates that the configuration is evolving too slowly because candidate points are too similar to the current location. The update formula used by Corana, Marchesi, Martini, and Ridella is:

$$v_i = \begin{cases} v_i\left(1 + c_i \frac{a_i/n_s - 0.6}{0.4}\right) & \text{if } a_i > 0.6 n_s \\ v_i\left(1 + c_i \frac{0.4 - a_i/n_s}{0.4}\right)^{-1} & \text{if } a_i < 0.4 n_s \\ v_i & \text{otherwise} \end{cases} \tag{8.8}$$

The c_i parameter controls the step variation along each direction and is typically set to 2 as shown in figure 8.4. Algorithm 8.6 implements this update. The temperature is reduced every n_t step adjustments.

[17] A. Corana, M. Marchesi, C. Martini, and S. Ridella, "Minimizing Multimodal Functions of Continuous Variables with the 'Simulated Annealing' Algorithm," *ACM Transactions on Mathematical Software*, vol. 13, no. 3, pp. 262–280, 1987.

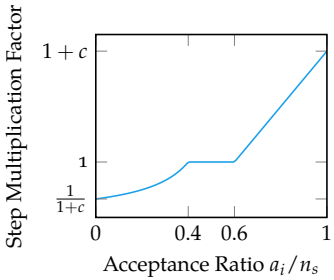


Figure 8.4. The step multiplication factor as a function of acceptance rate for $c = 2$.

```
function corana_update!(v, a, c, ns)
    for i in eachindex(v)
        ai, ci = a[i], c[i]
        if ai > 0.6ns
            v[i] *= (1 + ci*(ai/ns - 0.6)/0.4)
        elseif ai < 0.4ns
            v[i] /= (1 + ci*(0.4-ai/ns)/0.4)
        end
    end
    return v
end
```

Algorithm 8.6. The update formula used by Corana, Marchesi, Martini, and Ridella in adaptive simulated annealing, where `v` is a vector of coordinate step sizes, `a` is a vector of the number of accepted steps in each coordinate direction, `c` is a vector of step scaling factors for each coordinate direction, and `ns` is the number of cycles before running the step size adjustment.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The process is terminated when the temperature sinks low enough such that improvement can no longer be expected. Termination occurs when the most recent function value is no farther than ϵ from the previous n_ϵ iterations and the best function value obtained over the course of execution. Algorithm 8.7 provides an implementation and the algorithm is visualized in figure 8.5.

8.5 Cross-Entropy Method

The *cross-entropy method*,[18] in contrast with the methods we have discussed so far in this chapter, maintains an explicit probability distribution over the design space.[19] This probability distribution, often called a *proposal distribution*, is used to propose new samples for the next iteration. At each iteration, we sample from the proposal distribution and then update the proposal distribution to fit a fixed number of best samples, known as *elite samples*. The aim at convergence is for the proposal distribution to focus on the global optima. Algorithm 8.8 provides an implementation.

The cross-entropy method requires choosing a family of distributions parameterized by θ. One common choice is the family of multivariate normal distributions parameterized by a mean vector and a covariance matrix. The algorithm also requires us to specify the number of elite samples m_{elite} to use when fitting the parameters for the next iteration.

Depending on the choice of distribution family, the process of fitting the distribution to the elite samples can be done analytically. In the case of the multivariate normal distribution, the parameters are updated according to the maximum likelihood estimate:

$$\boldsymbol{\mu}^{(k+1)} = \frac{1}{m_{\text{elite}}} \sum_{i=1}^{m_{\text{elite}}} \mathbf{x}^{(i)} \tag{8.9}$$

$$\boldsymbol{\Sigma}^{(k+1)} = \frac{1}{m_{\text{elite}}} \sum_{i=1}^{m_{\text{elite}}} (\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k+1)})(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k+1)})^\top \tag{8.10}$$

Example 8.3 applies the cross-entropy method to a simple function. Figure 8.6 shows several iterations on a more complex function. Example 8.4 shows the potential limitation of using a multivariate normal distribution for fitting elite samples.

[18] R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Springer, 2004.

[19] The name of this method comes from the fact that the process of fitting the distribution involves minimizing *cross-entropy*, which is equivalent to minimizing the *Kullback–Leibler divergence*. Under certain conditions, minimizing the cross-entropy corresponds to finding the maximum likelihood estimate of the parameters of the distribution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function adaptive_simulated_annealing(f, x, v, t, ϵ;
    ns=20, nϵ=4, nt=max(100,5length(x)),
    γ=0.85, c=fill(2,length(x)) )

    y = f(x)
    best = (x=x, y=y)
    y_arr, n, U = [], length(x), Uniform(-1.0,1.0)
    a,counts_cycles,counts_resets = zeros(n), 0, 0

    while true
        for i in 1:n
            x′ = x + basis(i,n)*rand(U)*v[i]
            y′ = f(x′)
            Δy = y′ - y
            if Δy < 0 || rand() < exp(-Δy/t)
                x, y = x′, y′
                a[i] += 1
                if y′ < best.y; best = (x=x′, y=y′); end
            end
        end

        counts_cycles += 1
        counts_cycles ≥ ns || continue

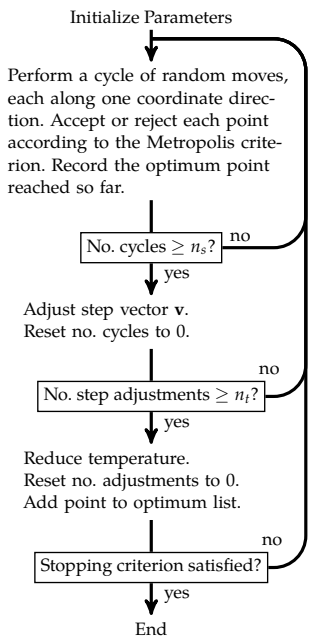
        counts_cycles = 0
        corana_update!(v, a, c, ns)
        fill!(a, 0)
        counts_resets += 1
        counts_resets ≥ nt || continue

        t *= γ
        counts_resets = 0
        push!(y_arr, y)

        if !(length(y_arr) > nϵ && y_arr[end] - best.y ≤ ϵ &&
             all(abs(y_arr[end]-y_arr[end-u]) ≤ ϵ for u in 1:nϵ))
            x, y = best
        else
            break
        end
    end
    return best.x
end
```

Algorithm 8.7. The adaptive simulated annealing algorithm, where `f` is the multivariate objective function, `x` is the starting point, `v` is the starting step vector, `t` is the starting temperature, and `ϵ` is the termination criterion parameter. The optional parameters are the number of cycles before running the step size adjustment `ns`, the number of step adjustments before reducing the temperature `nt`, the number of successive temperature reductions to test for termination `nϵ`, the temperature reduction coefficient `γ`, and the direction-wise varying criterion `c`.

Below is a flowchart for the adaptive simulated annealing algorithm as presented in the original paper.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

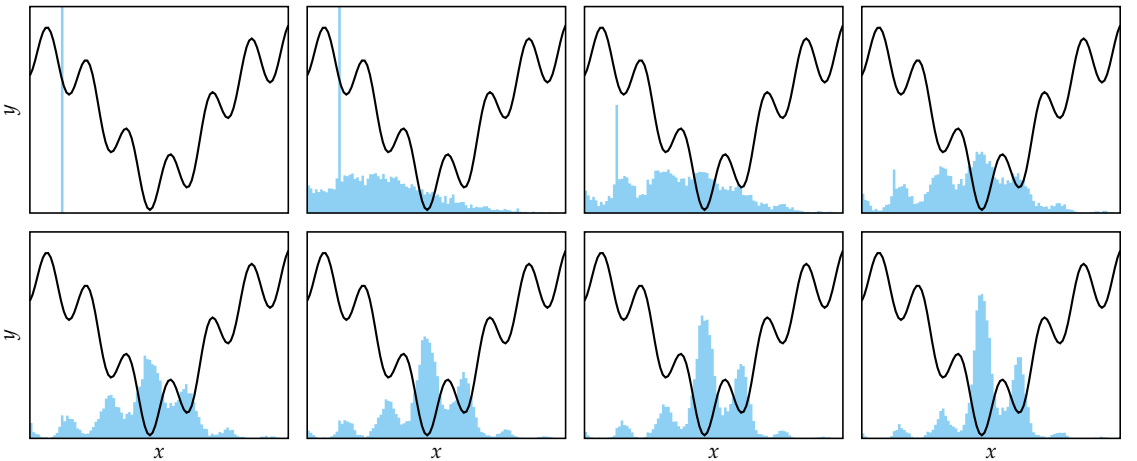


Figure 8.5. Simulated annealing with an exponentially decaying temperature, where the histograms indicate the probability of simulated annealing being at a particular position at that iteration.

```
using Distributions
function cross_entropy_method(f, P, k_max, m=100, m_elite=10)
    for k in 1 : k_max
        samples = rand(P, m)
        order = sortperm([f(samples[:,i]) for i in 1:m])
        P = fit(typeof(P), samples[:,order[1:m_elite]])
    end
    return P
end
```

Algorithm 8.8. The cross-entropy method, which takes an objective function `f` to be minimized, a proposal distribution `P`, an iteration count `k_max`, a sample size `m`, and the number of samples to use when refitting the distribution `m_elite`. It returns the updated distribution over where the global minimum is likely to exist.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

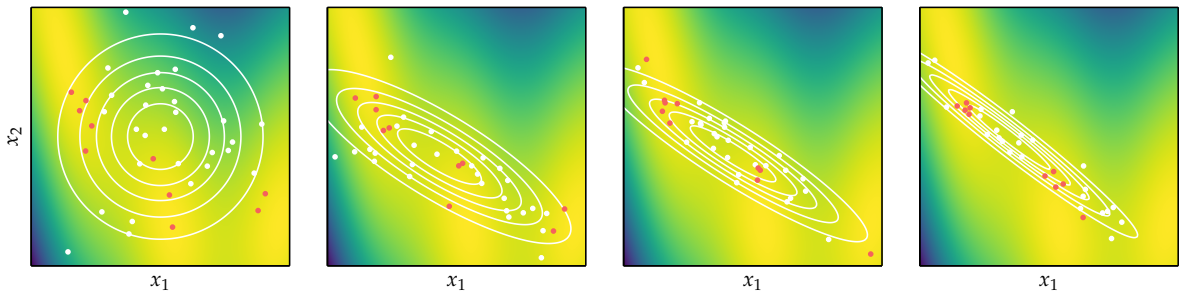
We can use `Distributions.jl` to represent, sample from, and fit proposal distributions. The parameter vector θ is replaced by a distribution `P`. Calling `rand(P,m)` will produce an $n \times m$ matrix corresponding to m samples of dimension n from `P`, and calling `fit` will fit a new distribution of the given input type.

```
import Random: seed!
import LinearAlgebra: norm
import Distributions: MvNormal
seed!(0) # set random seed for reproducible results
f = x->norm(x)
μ = [0.5, 1.5]
Σ = [1.0 0.2; 0.2 2.0]
P = MvNormal(μ, Σ)
k_max = 10
P = cross_entropy_method(f, P, k_max)
@show P.μ
```

```
P.μ = [-1.7329565477016673e-6, -7.042440586544702e-7]
```

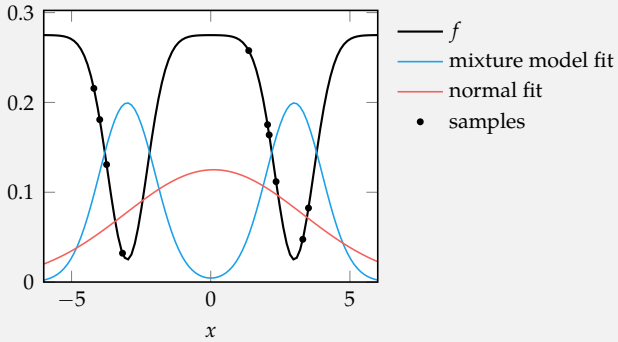
Example 8.3. An example of using the cross-entropy method.

Figure 8.6. The cross-entropy method with $m = 40$ applied to the Branin function (appendix B.3) using a multivariate Gaussian proposal distribution. The 10 elite samples in each iteration are in red.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The distribution family should be flexible enough to capture the relevant features of the objective function. Here we show the limitations of using a normal distribution on a multimodal objective function, which assigns greater density in between the two minima. A mixture model is able to center itself over each minimum.



Example 8.4. The normal distribution is unable to capture multiple local minima, in contrast to mixture models which can maintain several.

8.6 Proposal Distribution Descent

Like the cross-entropy method, *proposal distribution descent*[20] optimizes a proposal distribution parameterized by θ. We have to specify the proposal distribution family and the number of samples. The aim is to minimize the expectation $\mathbb{E}_{\mathbf{x}\sim p(\cdot\mid\theta)}[f(\mathbf{x})]$. Instead of fitting elite samples, the distribution is improved with gradient descent. The gradient is estimated from the samples:[21]

$$\begin{aligned}
\nabla_\theta \, \mathbb{E}_{\mathbf{x}\sim p(\cdot\mid\theta)}[f(\mathbf{x})] &= \int \nabla_\theta p(\mathbf{x} \mid \theta) f(\mathbf{x}) \, d\mathbf{x} && (8.11)\\
&= \int \frac{p(\mathbf{x} \mid \theta)}{p(\mathbf{x} \mid \theta)} \nabla_\theta p(\mathbf{x} \mid \theta) f(\mathbf{x}) \, d\mathbf{x} && (8.12)\\
&= \int p(\mathbf{x} \mid \theta) \nabla_\theta \log p(\mathbf{x} \mid \theta) f(\mathbf{x}) \, d\mathbf{x} && (8.13)\\
&= \mathbb{E}_{\mathbf{x}\sim p(\cdot\mid\theta)}[f(\mathbf{x}) \nabla_\theta \log p(\mathbf{x} \mid \theta)] && (8.14)\\
&\approx \frac{1}{m} \sum_{i=1}^{m} f(\mathbf{x}^{(i)}) \nabla_\theta \log p(\mathbf{x}^{(i)} \mid \theta) && (8.15)
\end{aligned}$$

Although we do not need the gradient of the objective function, we do need the gradient of the log likelihood, $\log p(\mathbf{x} \mid \theta)$. Example 8.5 shows how to compute

[20] The method discussed here is the same as using "vanilla" gradients for evolution strategies in D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, no. 15, pp. 949–980, 2014.

[21] This gradient estimation has been successfully applied to proposal distributions represented by deep neural networks. T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

the gradient of the log likelihood for the multivariate normal distribution. The estimated gradient can be used along with any of the descent methods discussed in previous chapters to improve θ. Algorithm 8.9 uses gradient descent with a fixed step factor. Figure 8.7 shows a few iterations of the algorithm.

```
using Distributions
function proposal_distribution_descent(f, θ, k_max; m=100, α=0.01)
    for k in 1:k_max
        samples = rand(θ, m)
        θ -= α*sum(f(x)*∇logp(x, θ) for x in samples)/m
    end
    return θ
end
```

Algorithm 8.9. Proposal distribution descent, which takes an objective function `f`, initial distribution parameter vector `θ`, iteration count `k_max`, sample size `m`, and step factor `α`. An optimized parameter vector is returned. The method `rand(θ, m)` generates `m` samples from the distribution parameterized by `θ`, and `∇logp(x, θ)` returns the log likelihood gradient.

8.7 Information-Geometric Optimization

Information-geometric optimization[22] builds on proposal distribution descent from section 8.6. The proposal distribution's parameters θ are iteratively improved using

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha \tilde{\nabla}_\theta \, \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \theta)}[f(\mathbf{x})] \tag{8.16}$$

where α is a step factor and $\tilde{\nabla}_\theta$ is what is called the *natural gradient*.[23] Using the natural gradient makes this algorithm invariant to changes in the parameterization of the distribution. In other words, reparameterizing the distribution will not affect the optimization trajectory.

The natural gradient $\tilde{\nabla}_\theta$ is defined in terms of the *Fisher information matrix*:

$$\mathbf{F}(\theta) = \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \theta)}\left[\nabla_\theta \log p(\mathbf{x} \mid \theta) \nabla_\theta \log p(\mathbf{x} \mid \theta)^\top\right] \tag{8.17}$$

The Fisher information matrix measures how much information an observation of $\mathbf{x}$ carries about an unknown parameter θ upon which the probability of $\mathbf{x}$ depends.[24] The natural gradient is computed by premultiplying the standard gradient by the inverse of the Fisher information matrix:

$$\tilde{\nabla}_\theta \, \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \theta)}[f(\mathbf{x})] = \mathbf{F}(\theta)^{-1} \nabla_\theta \, \mathbb{E}_{\mathbf{x} \sim p(\cdot \mid \theta)}[f(\mathbf{x})] \tag{8.18}$$

[22] Y. Ollivier, L. Arnold, A. Auger, and N. Hansen, "Information-Geometric Optimization Algorithms: A Unifying Picture via Invariance Principles," *Journal of Machine Learning Research*, vol. 18, no. 18, pp. 1–65, 2017.

[23] The (non-natural) gradient points in the direction that for small steps achieves the largest change in the objective for a given L_2 distance in its parameter space. The natural gradient points in the direction of steepest ascent under a local quadratic approximation of the *Kullback-Leibler divergence*, a measure of divergence between distributions. Proposal distribution descent with the natural gradient is called *natural evolution strategies*. D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, no. 15, pp. 949–980, 2014.

[24] For greater intuition about the Fisher information matrix, see J. Martens, *Journal of Machine Learning Research*, vol. 21, pp. 1–76, 2020.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 8.5. A derivation of the log likelihood gradient equations for the multivariate Gaussian distribution. For the original derivation and several more sophisticated solutions for handling the positive definite covariance matrix, see D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, no. 15, pp. 949–980, 2014.

The multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ is a popular distribution family due to having analytic solutions. The likelihood in n dimensions has the form

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{n}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. The log likelihood is

$$\log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\left(n \log 2\pi + \log|\boldsymbol{\Sigma}| + (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$$

The parameters can be updated using their log likelihood gradients:

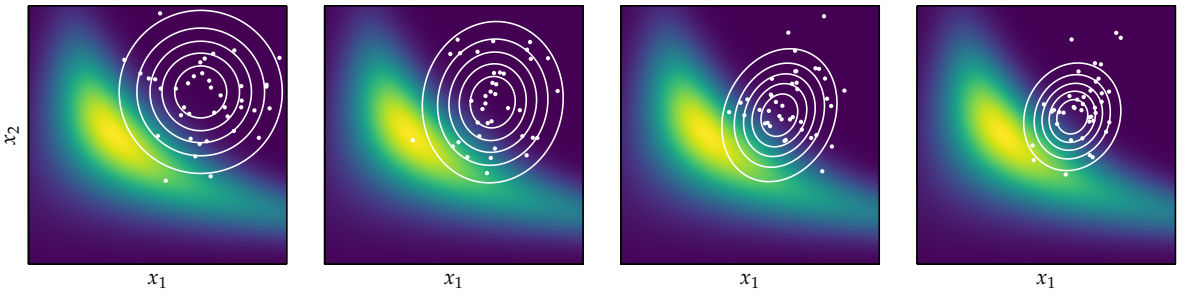
$$\nabla_{\boldsymbol{\mu}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})$$

$$\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2}\left(\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \boldsymbol{\Sigma}^{-1}\right)$$

Directly updating $\boldsymbol{\Sigma}$ may not result in a positive definite matrix, as is required for covariance matrices. One solution is to represent $\boldsymbol{\Sigma}$ as a product $\mathbf{A}^\top \mathbf{A}$, which guarantees that $\boldsymbol{\Sigma}$ remains positive semidefinite, and then update $\mathbf{A}$ instead. Replacing $\boldsymbol{\Sigma}$ by $\mathbf{A}^\top \mathbf{A}$ and taking the gradient with respect to $\mathbf{A}$ yields:

$$\nabla_{\mathbf{A}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{A}) = \mathbf{A}\left[\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})^\top\right]$$

Figure 8.7. Proposal distribution descent using multivariate normal distributions applied to Wheeler's Ridge (appendix B.7).



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Although we can use other parameterized distributions, we will derive the natural gradient updates for the multivariate normal distribution. From equation (8.15), we know that

$$\nabla_\theta \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\theta)}[f(\mathbf{x})] = \mathbb{E}_{\mathbf{x}\sim p(\cdot|\theta)}[f(\mathbf{x}) \nabla_\theta \log p(\mathbf{x} \mid \theta)] \tag{8.19}$$

For a multivariate normal $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, the log-density is

$$\log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\left(n \log 2\pi + \log|\boldsymbol{\Sigma}| + (\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \tag{8.20}$$

The gradients with respect to the mean and covariance are:

$$\nabla_{\boldsymbol{\mu}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \tag{8.21}$$

$$\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2}\left(\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \boldsymbol{\Sigma}^{-1}\right) \tag{8.22}$$

The (non-natural) gradients of the expected value of $f(\mathbf{x})$ are:

$$\nabla_{\boldsymbol{\mu}} \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}[f(\mathbf{x})] = \quad \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}\left[f(\mathbf{x})\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right] \tag{8.23}$$

$$\nabla_{\boldsymbol{\Sigma}} \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}[f(\mathbf{x})] = \frac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}\left[f(\mathbf{x})\left(\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \boldsymbol{\Sigma}^{-1}\right)\right] \tag{8.24}$$

From these equations we see that updates to the distribution depend on $\boldsymbol{\Sigma}^{-1}$, which can be affected by scale and may not remain invertible. Conveniently, the Fisher information matrix for the multivariate normal distribution has an analytic solution, and its inverse removes the dependence on $\boldsymbol{\Sigma}^{-1}$ in the above gradients, resulting in

$$\tilde{\nabla}_{\boldsymbol{\mu}} \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}[f(\mathbf{x})] = \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}[f(\mathbf{x})(\mathbf{x} - \boldsymbol{\mu})] \tag{8.25}$$

$$\tilde{\nabla}_{\boldsymbol{\Sigma}} \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}[f(\mathbf{x})] = \frac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu},\boldsymbol{\Sigma})}\left[f(\mathbf{x})\left((\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top - \boldsymbol{\Sigma}\right)\right] \tag{8.26}$$

The update for the mean is then

$$\boldsymbol{\mu}^{(k+1)} \leftarrow \boldsymbol{\mu}^{(k)} - \alpha \, \mathbb{E}_{\mathbf{x}\sim p(\cdot|\boldsymbol{\mu}^{(k)},\boldsymbol{\Sigma}^{(k)})}\left[f(\mathbf{x})(\mathbf{x} - \boldsymbol{\mu}^{(k)})\right] \tag{8.27}$$

We approximate this expectation using m samples where $\mathbf{x}^{(i)} \sim p(\cdot \mid \boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)})$:

$$\boldsymbol{\mu}^{(k+1)} \leftarrow \boldsymbol{\mu}^{(k)} - \alpha \frac{1}{m} \sum_{i=1}^{m} f(\mathbf{x}^{(i)})\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}\right) \tag{8.28}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In practice, directly using the objective values $f(\mathbf{x}^{(i)})$ in the sample-based approximation can be undesirable. The scale of f affects the step size, outliers can dominate the update, and affine transformations of the objective can significantly alter the optimization trajectory. To address these issues, information-geometric optimization replaces the raw objective values with nonnegative weights w_i that depend only on the relative quality of the sampled designs. The samples are ranked from best to worst, and weights are assigned as a function of rank rather than absolute objective value, with better-performing samples receiving larger weights. This scheme preserves the ordering induced by the objective while making the update invariant to strictly increasing transformations that preserve the sample ranking. The weights are typically normalized so that $\sum_{i=1}^{m} w_i = 1$.

[25] For optimization in $\mathbb{R}^n$, it is recommended to use at least $m = 4 + \lfloor 3 \ln n \rfloor$ samples per iteration, and $m_{\text{elite}} = \lfloor m/2 \rfloor$ elite samples. A recommended weighting is

$$w_i \propto \ln(m_{\text{elite}} + 1) - \ln i \quad \text{for } i \text{ in } 1 : m_{\text{elite}}$$

The mean update becomes

$$\boldsymbol{\mu}^{(k+1)} \leftarrow \boldsymbol{\mu}^{(k)} + \alpha \sum_{i=1}^{m} w_i \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)} \right) \tag{8.29}$$

Using $w_i = 1/m_{\text{elite}}$ for the first m_{elite} samples, having the remaining weights being 0, and using $\alpha = 1$ recovers the mean update in the cross-entropy method. Recommended weightings instead distribute greater weight to better designs.[25]

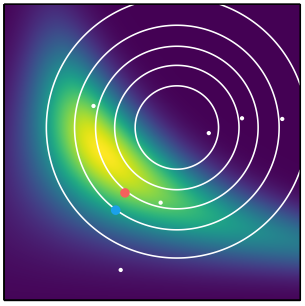


Figure 8.8. An initial proposal distribution (white contours), six samples (white dots), and the new updated means for both information-geometric optimization (blue dot) and the cross-entropy method (red dot) using three elite samples and $\alpha = 1$. Information-geometric optimization tends to update the mean more aggressively than the cross-entropy method, as it assigns higher weight to better sampled designs.

The update for the covariance matrix under a natural gradient step when estimated by m design samples is

$$\boldsymbol{\Sigma}^{(k+1)} \leftarrow (1 - \alpha)\boldsymbol{\Sigma}^{(k)} + \alpha \sum_{i=1}^{m} w_i (\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)})(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)})^\top \tag{8.30}$$

This update is called a rank-m_{elite} update when the weights $\mathbf{w}$ are only nonzero for the first m_{elite} samples, causing the second term to be a sum of m_{elite} rank-1 matrices and thus overall at most rank-m_{elite}.

Figure 8.8 compares the mean updates for information-geometric optimization and the cross-entropy method. Algorithm 8.10 provides an implementation.

8.8 Covariance Matrix Adaptation

Another popular method is *covariance matrix adaptation*,[26] which is also referred to as *CMA-ES* or *covariance matrix adaptation evolutionary strategy*.[27] This method makes a number of changes to information-geometric optimization to improve efficiency and robustness.

[26] It is common to use the phrase evolution strategies to refer specifically to covariance matrix adaptation.

[27] N. Hansen, "The CMA Evolution Strategy: A Tutorial," 2016. arXiv: 1604.00772.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function information_geometric_optimization(f, μ, Σ, k_max;
    m = 4 + floor(Int, 3*log(length(μ))),
    m_elite = div(m,2),
    α = 1.0)

    ws = log(m_elite+1) .- log.(1:m_elite)
    normalize!(ws)
    ws = vcat(ws, zeros(m - m_elite))

    for k in 1:k_max
        P = MvNormal(μ, Σ)
        xs = [rand(P) for i in 1:m]
        ys = f.(xs)
        is = sortperm(ys) # best to worst

        # covariance adaptation
        Σ = (1-α)*Σ +
            α*sum(ws[i]*(xs[is[i]] - μ)*(xs[is[i]] - μ)' for i in 1:m)
        Σ = (Σ + Σ')/2 # enforce symmetry

        # mean update
        μ += α*sum(ws[i]*(xs[is[i]] - μ) for i in 1:m)
    end
    return (μ, Σ)
end
```

Algorithm 8.10. Information-geometric optimization using multivariate Gaussian distributions, which takes an objective function `f`, initial normal distribution with mean `μ` and covariance `Σ`, and iteration count `k_max`. We can optionally specify the sample size `m`, number of elite samples `m_elite`, and step factor `α`. The covariance matrix is symmetrized at the end. The final mean of the proposal distribution is returned as an estimate of the minimizer.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

First, rather than operating on multivariate Gaussians of the form $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, covariance matrix adaptation uses $\mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{C})$, splitting the covariance into a matrix $\mathbf{C}$ that controls the shape of the distribution, and a scalar σ that controls the scale of the distribution. This parameterization allows the scale of the distribution to update more rapidly than the shape.

Following the same derivation for these parameters, we get:

$$\mathbf{C}^{(k+1)} \leftarrow \left(1 - \alpha_e \sum_{i=1}^{m} w_i\right)\mathbf{C}^{(k)} + \alpha_e \sum_{i=1}^{m} w_i \left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)^\top \tag{8.31}$$

where α_e is a step factor with a distinguishing subscript because covariance matrix adaptation uses multiple step factors. This update is a rank-m_{elite} update when the weights $\mathbf{w}$ are only nonzero for the first m_{elite} samples.[28] Small values of α_e result in slow learning. Large values of α_e can lead to degenerate covariance matrices. A recommended step factor is $\alpha_e \approx m_{\text{eff}}/n^2$, where m_{eff} is the effective sample size:

$$m_{\text{eff}} = \frac{1}{\sum_{i=1}^{m} w_i^2} \tag{8.32}$$

[28] Figure 6 in the original paper uses a more complicated formulation that allows both positive and negative weights that need not sum to one, resulting in more work to ensure this update remains rank-m_{elite}. For the implementation here, the weights sum to one and only the first m_{elite} weights are nonzero.

The rank-m_{elite} update only incorporates information from the current population. A form of momentum is incorporated by also updating the covariance matrix based on how the mean changes over time. Without momentum, such a rank-1 update has the form:

$$\mathbf{C}^{(k+1)} \leftarrow (1 - \alpha_1)\mathbf{C}^{(k)} + \alpha_1 \left(\frac{\boldsymbol{\mu}^{(k+1)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)\left(\frac{\boldsymbol{\mu}^{(k+1)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)^\top \tag{8.33}$$

Momentum is introduced using exponential smoothing:

$$\mathbf{C}^{(k+1)} \leftarrow (1 - \alpha_1)\mathbf{C}^{(k)} + \alpha_1 \left(\mathbf{p}_c^{(k+1)}\right)\left(\mathbf{p}_c^{(k+1)}\right)^\top \tag{8.34}$$

where $\mathbf{p}_c^{(k+1)}$ is known as an *evolution path*:[29]

$$\mathbf{p}_c^{(1)} = \mathbf{0} \tag{8.35}$$

$$\mathbf{p}_c^{(k+1)} \leftarrow (1 - \alpha_c)\mathbf{p}_c^{(k)} + \sqrt{\alpha_c(2 - \alpha_c)m_{\text{eff}}}\left(\frac{\boldsymbol{\mu}^{(k+1)} - \boldsymbol{\mu}^{(k)}}{\alpha_\mu \sigma^{(k)}}\right) \tag{8.36}$$

[29] The factor $\sqrt{\alpha_c(2-\alpha_c)m_{\text{eff}}}$ ensures that $(1-\alpha_c)^2 + \sqrt{\alpha_c(2-\alpha_c)}^2 = 1$, which prevents excessive growth or shrinkage.

where α_c is the covariance evolution path step factor and α_μ is the step factor for the mean update.[30]

[30] It is recommended that $\alpha_1 \approx 2/n^2$, $\alpha_c \leq 1$, and $1/\alpha_c \in [\sqrt{n}, n]$.

The overall covariance update is a combination of these updates:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\mathbf{C}^{(k+1)} \leftarrow (1-\alpha_1-\alpha_e)\mathbf{C}^{(k)} + \alpha_1\left(\mathbf{p}_c^{(k+1)}\right)\left(\mathbf{p}_c^{(k+1)}\right)^\top + \alpha_e \sum_{i=1}^{m} w_i \left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)\left(\frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k)}}{\sigma^{(k)}}\right)^\top \tag{8.37}$$

The scale parameter σ is also updated using an evolution path:

$$\mathbf{p}_\sigma^{(1)} = \mathbf{0} \tag{8.38}$$

$$\mathbf{p}_\sigma^{(k+1)} \leftarrow (1-\alpha_\sigma)\mathbf{p}_\sigma^{(k)} + \sqrt{\alpha_\sigma(2-\alpha_\sigma)m_{\text{eff}}}\left(\mathbf{C}^{(k)}\right)^{-\frac{1}{2}}\left(\frac{\boldsymbol{\mu}^{(k+1)} - \boldsymbol{\mu}^{(k)}}{\alpha_\mu \sigma^{(k)}}\right) \tag{8.39}$$

where α_σ is the scale evolution path step factor.[31] The matrix multiplication rescales the step into an orthonormal basis so the evolution path vector can be compared to samples drawn from a unit multivariate Gaussian.

[31] It is also recommended that $1/\alpha_\sigma \in [\sqrt{n}, n]$.

Intuitively, as depicted in figure 8.9, when successive steps tend to double back and generally remain in the same vicinity, the scale should decrease. When successive steps are correlated and consistent progress is made, the scale should increase.

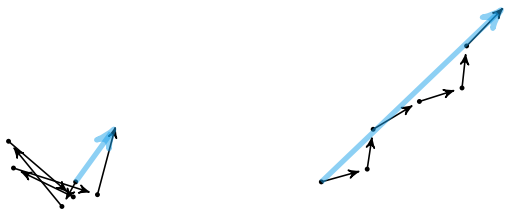


Figure 8.9. Successive mean updates for covariance matrix adaptation in black, with the overall trend captured by an evolution path shown in blue. On the left, the updates remain roughly centered around the same location, so the scale should be reduced. On the right, the updates make progress, so the scale should be increased.

Ideally, the length of the evolution path $\mathbf{p}_\sigma$ is equal to the expected length under random selection. If the length is greater than expected, we increase the scale. If the length is lower than expected, we decrease the scale. We do this with the update:

$$\sigma^{(k+1)} \leftarrow \sigma^{(k)} \exp\left(\frac{\alpha_\sigma}{\beta_\sigma}\left(\frac{\left\|\mathbf{p}_\sigma^{(k+1)}\right\|}{\mathbb{E}_{\mathbf{x}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[\|\mathbf{x}\|]} - 1\right)\right) \tag{8.40}$$

where $\beta_\sigma \approx 1$ is a damping factor and the expected length of a vector sampled from a unit multivariate Gaussian is:

$$\mathbb{E}_{\mathbf{x}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[\|\mathbf{x}\|] = \frac{\sqrt{2}\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{n}{2}\right)} \tag{8.41}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Covariance matrix adaptation is illustrated in figure 8.10 and implemented in algorithm 8.11.

8.9 Summary

- Stochastic methods employ random numbers during the optimization process.
- Mesh adaptive direct search is a pattern search method that uses random patterns.
- The zero-order stochastic step is a random step used for optimizing very large objective functions that can be computed in-place using two objective evaluations and a pseudo-random number generator.
- Simulated annealing uses a temperature that controls random exploration and that is reduced over time to converge on a local minimum.
- The cross-entropy method and evolution strategies maintain proposal distributions from which they sample in order to inform updates.
- Proposal distribution descent uses gradient descent with respect to the log likelihood to update its proposal distribution.
- Information-geometric optimization uses natural gradient updates to be invariant to how its distribution is parameterized, and only uses the fitness ranking in order to be invariant to order-preserving transforms of the objective.
- Covariance matrix adaptation extends information-geometric optimization with independent shape and scale parameters for the covariance matrix and employs evolution paths to improve learning.

8.10 Exercises

Exercise 8.1. We have shown that mixture proposal distributions can better capture multiple minima. Why might their use in the cross-entropy method be limited?

Solution: The cross-entropy method must fit distribution parameters with every iteration. Unfortunately, no known analytic solutions for fitting multivariate mixture distributions exist. Instead, one commonly uses the iterative expectation maximization algorithm to converge on an answer.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function covariance_matrix_adaptation_weights(m, m_elite)
    ws = log(m_elite+1) .- log.(1:m_elite)
    normalize!(ws)
    return vcat(ws, zeros(m - m_elite))
end

function covariance_matrix_adaptation(f, μ, σ, C, k_max;
    n = length(μ),
    m = 4 + floor(Int, 3*log(n)),
    m_elite = div(m,2), # number of elite samples
    ws = covariance_matrix_adaptation_weights(m, m_elite),
    m_eff = 1/sum(ws.^2),
    α1 = 2/n^2,         # covariance rank-1 step factor
    αe = m_eff/n^2,     # covariance rank-m elite step factor
    αc = 1/sqrt(n),     # covariance evo path step factor
    αμ = 1.0 - α1,      # mean step factor
    ασ = 1/sqrt(n),     # scale step factor
    βσ = 1.0            # scale damping
    )

    @assert 1-α1-αe > 0
    expected_len = sqrt(2)*gamma((n+1)/2)/gamma(n/2)
    pσ, pc = zeros(n), zeros(n)

    for k in 1:k_max
        P = MvNormal(μ, σ^2*C)
        xs = [rand(P) for i in 1:m]
        ys = [f(x) for x in xs]
        is = sortperm(ys) # best to worst

        # mean update
        μ′ = μ + αμ * sum(ws[i]*(xs[is[i]] - μ) for i in 1:m)

        # step-size control
        δ = (μ′ - μ)/(αμ*σ)
        pσ = (1-ασ)*pσ + sqrt(ασ*(2-ασ)*m_eff)*C^-0.5*δ
        σ *= exp(ασ/βσ * (norm(pσ)/expected_len - 1))

        # covariance adaptation
        pc = (1-αc)*pc + sqrt(αc*(2-αc)*m_eff)*δ
        C = (1-α1-αe)*C +
            α1*pc*pc' +
            αe/σ^2*sum(ws[i]*(xs[is[i]]-μ)*(xs[is[i]]-μ)' for i in 1:m)
        C = (C + C')/2 # enforce symmetry

        μ = μ′
    end
    return (μ, σ, C)
end
```

Algorithm 8.11. Covariance matrix adaptation, which takes an objective function `f` to be minimized; an initial distribution mean `μ`, scale `σ`, and shape `C`; and an iteration count `k_max`. One can optionally specify the sample size `m`, the number of elite samples `m_elite`, and various step factors.

The `gamma` function is provided by `SpecialFunctions.jl`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

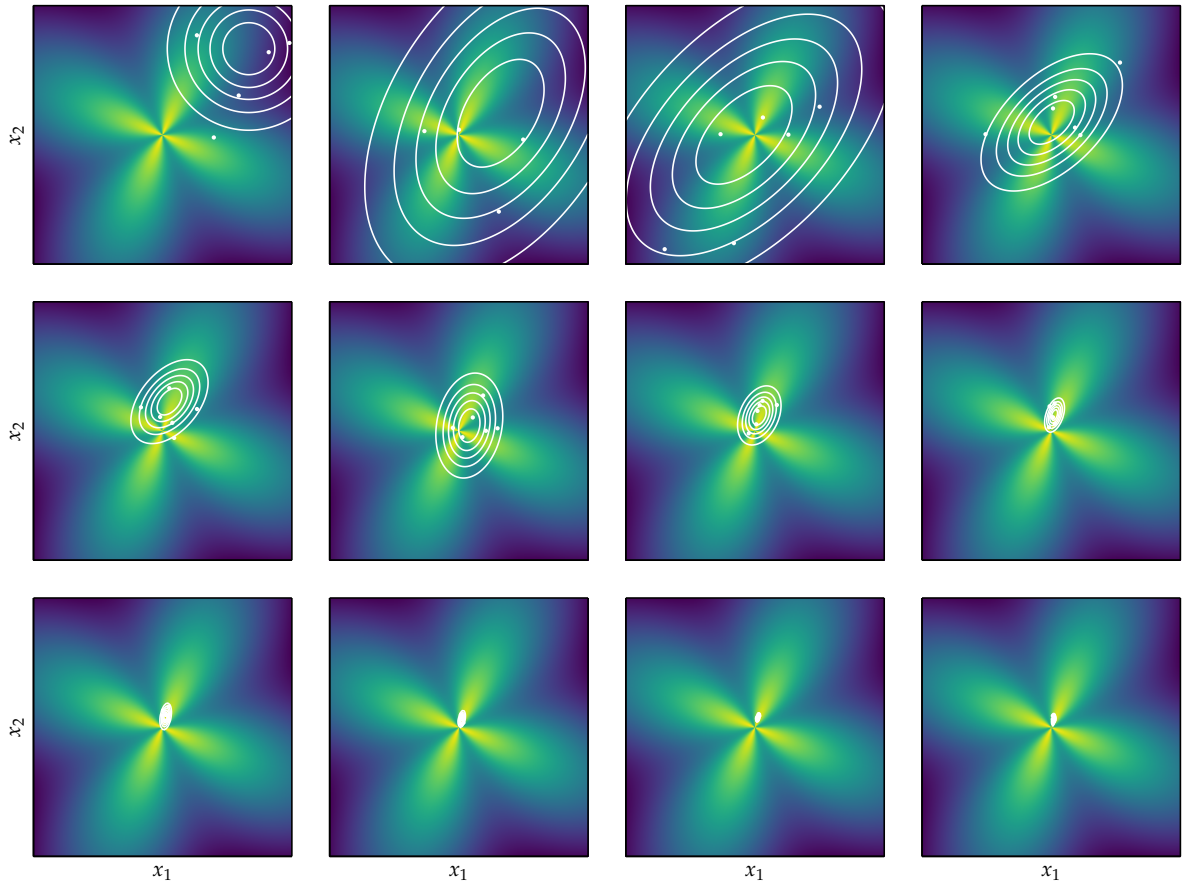


Figure 8.10. Covariance matrix adaptation using multivariate Gaussian distributions applied to the flower function (appendix B.4).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 8.2. In the cross-entropy method, what is a potential effect of using an elite sample size that is very close to the total sample size?

Solution: If the number of elite samples is close to the total number of samples, then the resulting distribution will closely match the population. There will not be a significant bias toward the best locations for a minimizer, and so convergence will be slow.

Exercise 8.3. Consider proposal distribution descent with a Gaussian distribution $\mathcal{N}(\mu, \nu)$, with the mean on the optimum, $\mu = x^*$. Show that the update for the variance ν remains constant (does not vanish) as ν approaches zero, and explain why a fixed step factor α poses a numerical risk.

Solution: The derivative used in the descent update for the variance is:

$$\nabla_\nu \, \mathbb{E}_{x\sim\mathcal{N}(\mu,\nu)}[f(x)] \approx \frac{1}{m}\sum_{i=1}^{m} f(x^{(i)}) \frac{\partial}{\partial \nu} \log \mathcal{N}(x^{(i)} \mid \mu, \nu)$$

where

$$\begin{aligned}\frac{\partial}{\partial \nu} \log \mathcal{N}(x^{(i)} \mid \mu, \nu) &= \frac{\partial}{\partial \nu}\left(-\frac{1}{2}\log 2\pi - \frac{1}{2}\log \nu - \frac{(x^{(i)} - \mu)^2}{2\nu}\right) \\ &= -\frac{1}{2\nu} + \frac{(x^{(i)} - \mu)^2}{2\nu^2}\end{aligned}$$

Under maximum likelihood estimation, the expected gradient with respect to samples drawn from a distribution is zero:

$$\mathbb{E}_{x\sim\mathcal{N}(\mu,\nu)}\left[(x-\mu)^2\right] = \nu \quad \Rightarrow \quad \mathbb{E}_{x\sim\mathcal{N}(\mu,\nu)}\left[\frac{\partial}{\partial \nu}\ell\right] = -\frac{1}{2\nu} + \frac{1}{2\nu} = 0$$

However, in proposal distribution descent, gradient samples are weighted according to their objective function values. Assume that f is twice differentiable and has a strict local minimum at $\mu = x^*$. Locally, we may approximate:

$$f(x) \approx \frac{1}{2}h(x-\mu)^2 \qquad h = f''(\mu) > 0$$

Substituting this approximation yields

$$\nabla_\nu \, \mathbb{E}_{x\sim\mathcal{N}(\mu,\nu)}[f(x)] \approx \mathbb{E}_{x\sim\mathcal{N}(\mu,\nu)}\left[\frac{1}{2}h(x-\mu)^2\left(-\frac{1}{2\nu} + \frac{(x-\mu)^2}{2\nu^2}\right)\right]$$

Letting $z = x - \mu \sim \mathcal{N}(0, \nu)$ and using $\mathbb{E}[z^2] = \nu$ and $\mathbb{E}[z^4] = 3\nu^2$, we obtain

$$\nabla_\nu \, \mathbb{E}_{z\sim\mathcal{N}(0,\nu)}[f(z+\mu)] \approx \frac{h}{2}\left(-\frac{1}{2\nu}\,\mathbb{E}[z^2] + \frac{1}{2\nu^2}\,\mathbb{E}[z^4]\right) = \frac{h}{2} > 0$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Thus, when the mean is located at the optimum, the expected gradient with respect to the variance is positive, and a gradient descent update $\nu \leftarrow \nu - \alpha \nabla_\nu$ decreases the variance.

Although the expected gradient remains bounded as $\nu \to 0$, the $\partial_\nu \log \mathcal{N}(x \mid \mu, \nu)$ term scales as $O(1/\nu)$. As a result, the Monte Carlo gradient estimate can exhibit large fluctuations when ν is small. For any fixed step factor α, there exists a sufficiently small ν such that a single update causes the variance to become negative. This can be mitigated in practice with step-size control, explicit lower bounds on ν, or reparameterizations (e.g., $\nu = e^\eta$).

Exercise 8.4. Derive the maximum likelihood estimate for the cross-entropy method using multivariate normal distributions:

$$\boldsymbol{\mu}^{(k+1)} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}^{(i)}$$

$$\boldsymbol{\Sigma}^{(k+1)} = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k+1)})(\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(k+1)})^\top$$

where the maximum likelihood estimates are the parameter values that maximize the likelihood of sampling the individuals $\left\{\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\right\}$.

Solution: The probability density of a design $\mathbf{x}$ under a multivariate normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ is

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{n}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

We can simplify the problem by maximizing the log-likelihood instead.[32] The log-likelihood is:

$$\begin{aligned} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= -\frac{1}{2} \log\left((2\pi)^n |\boldsymbol{\Sigma}|\right) - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \\ &= -\frac{1}{2} \log\left((2\pi)^n |\boldsymbol{\Sigma}|\right) - \frac{1}{2}\left(\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \mathbf{x} - 2\mathbf{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} + \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}\right) \end{aligned}$$

[32] The log function is strictly increasing for positive inputs, so maximizing $\log f(x)$ also maximizes a strictly positive $f(x)$.

We begin by maximizing the log-likelihood of the m individuals with respect to the mean:

$$\begin{aligned} \ell\left(\boldsymbol{\mu} \mid \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\right) &= \sum_{i=1}^{m} \log p(\mathbf{x}^{(i)} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \\ &= \sum_{i=1}^{m} -\frac{1}{2} \log\left((2\pi)^n |\boldsymbol{\Sigma}|\right) - \frac{1}{2}\left(\left(\mathbf{x}^{(i)}\right)^\top \boldsymbol{\Sigma}^{-1} \mathbf{x}^{(i)} - 2\left(\mathbf{x}^{(i)}\right)^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} + \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}\right) \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We compute the gradient using the facts that $\nabla_{\mathbf{z}} \mathbf{z}^\top \mathbf{A}\mathbf{z} = \left(\mathbf{A} + \mathbf{A}^\top\right)\mathbf{z}$, that $\nabla_{\mathbf{z}} \mathbf{a}^\top \mathbf{z} = \mathbf{a}$, and that $\boldsymbol{\Sigma}$ is symmetric and positive definite, and thus $\boldsymbol{\Sigma}^{-1}$ is symmetric:

$$\begin{aligned}
\nabla_{\boldsymbol{\mu}} \ell\left(\boldsymbol{\mu} \mid \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\right) &= \sum_{i=1}^{m} -\frac{1}{2}\left(\nabla_{\boldsymbol{\mu}}\left(-2\left(\mathbf{x}^{(i)}\right)^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right) + \nabla_{\boldsymbol{\mu}}\left(\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right)\right) \\
&= \sum_{i=1}^{m}\left(\nabla_{\boldsymbol{\mu}}\left(\left(\mathbf{x}^{(i)}\right)^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right) - \frac{1}{2}\nabla_{\boldsymbol{\mu}}\left(\boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}\right)\right) \\
&= \sum_{i=1}^{m} \boldsymbol{\Sigma}^{-1}\mathbf{x}^{(i)} - \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}
\end{aligned}$$

We set the gradient to zero:

$$\begin{aligned}
\mathbf{0} &= \sum_{i=1}^{m} \boldsymbol{\Sigma}^{-1}\mathbf{x}^{(i)} - \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} \\
\sum_{i=1}^{m} \boldsymbol{\mu} &= \sum_{i=1}^{m} \mathbf{x}^{(i)} \\
m\boldsymbol{\mu} &= \sum_{i=1}^{m} \mathbf{x}^{(i)} \\
\boldsymbol{\mu} &= \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}^{(i)}
\end{aligned}$$

Next we maximize with respect to the inverse covariance, $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$, using the fact that $|\mathbf{A}^{-1}| = 1/|\mathbf{A}|$ with $\mathbf{b}^{(i)} = \mathbf{x}^{(i)} - \boldsymbol{\mu}$:

$$\begin{aligned}
\ell\left(\boldsymbol{\Lambda} \mid \boldsymbol{\mu}, \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\right) &= \sum_{i=1}^{m}\left(-\frac{1}{2}\log\left((2\pi)^n |\boldsymbol{\Lambda}|^{-1}\right) - \frac{1}{2}\left(\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}\right)^\top \boldsymbol{\Lambda}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}\right)\right)\right) \\
&= -\frac{mn}{2}\log(2\pi) + \sum_{i=1}^{m}\left(\frac{1}{2}\log(|\boldsymbol{\Lambda}|) - \frac{1}{2}\left(\mathbf{b}^{(i)}\right)^\top \boldsymbol{\Lambda}\mathbf{b}^{(i)}\right)
\end{aligned}$$

We compute the gradient using the facts that $\nabla_{\mathbf{A}}|\mathbf{A}| = |\mathbf{A}|\mathbf{A}^{-\top}$ and $\nabla_{\mathbf{A}} \mathbf{z}^\top \mathbf{A}\mathbf{z} = \mathbf{z}\mathbf{z}^\top$:

$$\begin{aligned}
\nabla_{\boldsymbol{\Lambda}} \ell\left(\boldsymbol{\Lambda} \mid \boldsymbol{\mu}, \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(m)}\right) &= \sum_{i=1}^{m} \nabla_{\boldsymbol{\Lambda}}\left(\frac{1}{2}\log(|\boldsymbol{\Lambda}|) - \frac{1}{2}\left(\mathbf{b}^{(i)}\right)^\top \boldsymbol{\Lambda}\mathbf{b}^{(i)}\right) \\
&= \sum_{i=1}^{m} \frac{1}{2|\boldsymbol{\Lambda}|}\nabla_{\boldsymbol{\Lambda}}|\boldsymbol{\Lambda}| - \frac{1}{2}\mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^\top \\
&= \sum_{i=1}^{m} \frac{1}{2|\boldsymbol{\Lambda}|}|\boldsymbol{\Lambda}|\boldsymbol{\Lambda}^{-\top} - \frac{1}{2}\mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^\top \\
&= \frac{1}{2}\sum_{i=1}^{m} \boldsymbol{\Lambda}^{-\top} - \mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^\top \\
&= \frac{1}{2}\sum_{i=1}^{m} \boldsymbol{\Sigma} - \mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^\top
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and set the gradient to zero:

$$\begin{aligned}
\mathbf{0} &= \frac{1}{2}\sum_{i=1}^{m} \boldsymbol{\Sigma} - \mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^{\top} \\
\sum_{i=1}^{m} \boldsymbol{\Sigma} &= \sum_{i=1}^{m} \mathbf{b}^{(i)}\left(\mathbf{b}^{(i)}\right)^{\top} \\
\boldsymbol{\Sigma} &= \frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}\right)\left(\mathbf{x}^{(i)} - \boldsymbol{\mu}\right)^{\top}
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

9 Population Methods

Previous chapters have focused on methods where a single design point is moved incrementally toward a minimum. This chapter presents a variety of *population methods* that involve optimization using a collection of design points, called *individuals*. Having a large number of individuals distributed throughout the design space can help the algorithm avoid becoming stuck in a local minimum.[1] Information at different points in the design space can be shared between individuals to globally optimize the objective function. Many population methods are stochastic in nature, and it is generally easy to parallelize the computation. Several methods that we discuss in this chapter are loosely inspired by evolutionary processes and other biological phenomena.

[1] If the structure of the problem is such that becoming stuck is not a concern, the local descent methods discussed earlier are often more efficient.

9.1 Population Iteration

In contrast with local descent methods (chapter 4), which iteratively improve a single design, population methods iteratively improve a population of m designs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$.[2] The population at a particular iteration is often referred to as a *generation*. The algorithms are designed so that the individuals in the population converge to one or more local minima over multiple generations. Population methods typically follow an iterative structure similar to local descent methods as shown in algorithm 9.1. The various methods discussed in this chapter differ in how they use the previous generation to generate the next generation.

[2] The superscript (i) denotes the ith individual in the population rather than the iteration number as in previous chapters.

Population methods begin with an *initial population*, just as descent methods require an initial design point. The initial population should be spread over the design space to increase the chances that the samples are close to the best regions. Some population methods require additional information to be associated with individuals, such as velocity in the case of particle swarm optimization.[3]

[3] Particle swarm optimization is discussed in section 9.4.

```
abstract type PopulationMethod end

function population_method(M::PopulationMethod, f, designs, k_max)
    population = init!(M, f, designs)
    for k in 1:k_max
        population = step!(M, f, population)
    end
    return population
end
```

Algorithm 9.1. An iterated population method for minimizing `f` starting with a population given by `designs`. This implementation operates on an abstract `PopulationMethod` object that supports both an `init!` method for initialization and a `step!` method for executing a single population step. It executes `k_max` iterations. The best performing individual can then be extracted from the population.

We can initialize the population by sampling from a variety of distributions. The choice of distribution can have a significant impact on the performance of the algorithm. The uniform distribution over a hyperrectangle is a common choice if the bounds on particular design variables are known. The multivariate normal distribution can be used to concentrate samples in areas of interest while also providing samples that broadly span the space. Alternatively, we can use the *Cauchy distribution*, which is a distribution that decays less rapidly than the Gaussian as we move away from the center. Figure 9.1 compares populations generated using different methods. More advanced sampling methods are discussed in chapter 16.

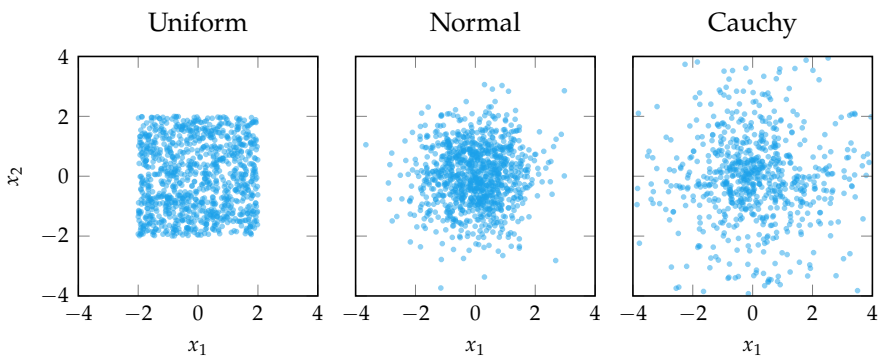


Figure 9.1. Initial populations of size 1,000 sampled using a uniform hyperrectangle bounded by ± 2, a zero-mean normal distribution with diagonal covariance $\mathbf{\Sigma} = \mathbf{I}$, and Cauchy distributions centered at the origin with scale $\sigma = 1$. The `Distributions.jl` package can be used to sample from a variety of distributions. M. Besançon, T. Papamarkou, D. Anthoff, A. Arslan, S. Byrne, D. Lin, et al., "Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem," *Journal of Statistical Software*, vol. 98, no. 16, pp. 1–30, 2021.

9.2 Genetic Algorithms

Genetic algorithms (algorithm 9.2) borrow inspiration from biological evolution,[4] where fitter individuals are more likely to pass on their genes to the next generation.[5] In this context, a *gene* corresponds to a design variable. An individual's

[4] In the biological context, DNA is evolved. DNA consists of a sequence of four nucleobases: adenine, thymine, cytosine, and guanine, which are often abbreviated A, T, C, and G. Genetic algorithms evolve individuals represented by assignments of values to design variables.

[5] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

fitness for reproduction is inversely related to the value of the objective function at that point. The design point associated with an individual is represented as a *chromosome*. At each generation, the chromosomes of the fitter individuals are passed on to the next generation after undergoing the genetic operations of *crossover* and *mutation*. There are some variations of genetic algorithms that incorporate *elitism* or *elitist selection*, where some of the best individuals are passed on to the next generation without undergoing these genetic operations.

```
struct GeneticAlgorithm <: PopulationMethod
    S # SelectionMethod
    C # CrossoverMethod
    U # MutationMethod
end

init!(M::GeneticAlgorithm, f, designs) = designs

function step!(M::GeneticAlgorithm, f, population)
    S, C, U = M.S, M.C, M.U
    parents = select(S, f.(population))
    children = [crossover(C,population[p[1]],population[p[2]])
                for p in parents]
    return [mutate(U, c) for c in children]
end
```

Algorithm 9.2. The genetic algorithm is defined by a selection method `S`, a crossover method `C`, and a mutation method `U`.

9.2.1 Selection

Selection is the process of choosing chromosomes to use as parents for the next generation. For a population with m chromosomes, a selection method will produce a list of m parental pairs[6] for the m children of the next generation. The selected pairs may contain duplicates.

[6] Alternatively, we can use groups if we want to combine more than two parents to form a child.

There are several approaches for biasing the selection toward the fittest (algorithm 9.3). In *truncation selection* (figure 9.2), we sample parents from among the best k chromosomes in the population. In *tournament selection* (figure 9.3), each parent is the fittest out of k randomly chosen chromosomes of the population. In *roulette wheel selection* (figure 9.4), also known as *fitness proportionate selection*, each parent is chosen with a probability proportional to its performance relative to the population. Since we are interested in minimizing an objective function f, the fitness of the ith individual $\mathbf{x}^{(i)}$ is inversely related to $y^{(i)} = f(\mathbf{x}^{(i)})$. There are differ-

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

ent ways to transform a collection $y^{(1)}, \ldots, y^{(m)}$ into fitnesses. A simple approach is to assign the fitness of individual i according to $\max\{y^{(1)}, \ldots, y^{(m)}\} - y^{(i)}$.

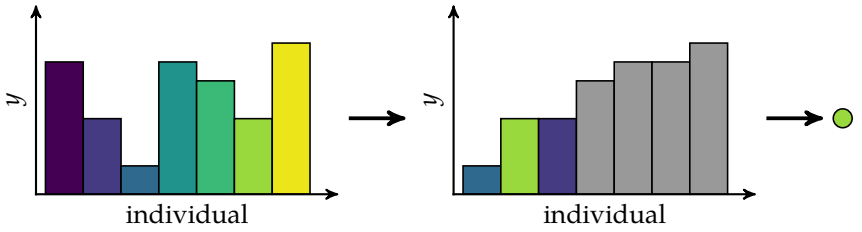


Figure 9.2. Truncation selection with a population size $m = 7$ and sample size $k = 3$. The height of a bar indicates its objective function value whereas its color indicates what individual it corresponds to.

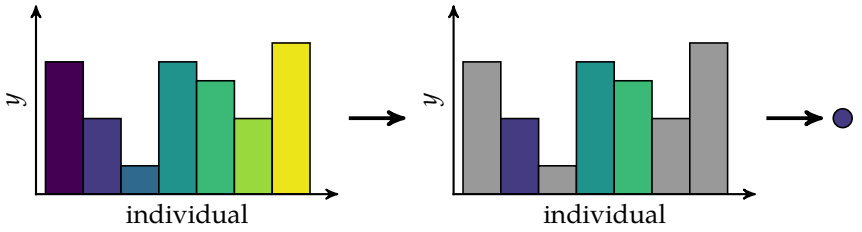


Figure 9.3. Tournament selection with a population size $m = 7$ and a sample size $k = 3$, which is run separately for each parent. The height of a bar indicates its objective function value, whereas its color indicates what individual it corresponds to.

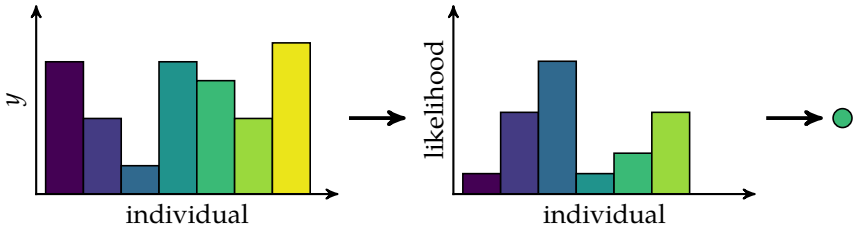


Figure 9.4. Roulette wheel selection with a population size $m = 7$, which is run separately for each parent. The approach used causes the individual with the worst objective function value to have a zero likelihood of being selected. The height of a bar indicates its objective function value (left), or its likelihood (right), whereas its color indicates what individual it corresponds to.

9.2.2 *Crossover*

Crossover combines the chromosomes of parents to form children. As with selection, there are several crossover schemes (algorithm 9.4).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
abstract type SelectionMethod end

# Pick pairs randomly from top k parents
struct TruncationSelection <: SelectionMethod
    k # top k to keep
end

function select(t::TruncationSelection, y)
    p = sortperm(y)
    return [p[rand(1:t.k, 2)] for i in y]
end

# Pick parents by choosing best among random subsets
struct TournamentSelection <: SelectionMethod
    k # tournament size
end

function select(t::TournamentSelection, y)
    getparent() = begin
        p = randperm(length(y))
        p[argmin(y[p[1:t.k]])]
    end
    return [[getparent(), getparent()] for i in y]
end

# Sample parents proportionately to fitness
struct RouletteWheelSelection <: SelectionMethod end

function select(::RouletteWheelSelection, y)
    y = maximum(y) .- y
    cat = Categorical(normalize(y, 1))
    return [rand(cat, 2) for i in y]
end
```

Algorithm 9.3. Several selection methods for genetic algorithms. Calling `select` with a `SelectionMethod` and a list of objective function values `y` will produce a list of parental pairs.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- In *single-point crossover* (figure 9.5), the first portion of parent A's chromosome forms the first portion of the child chromosome, and the latter portion of parent B's chromosome forms the latter part of the child chromosome. The *crossover point* where the transition occurs is determined uniformly at random.

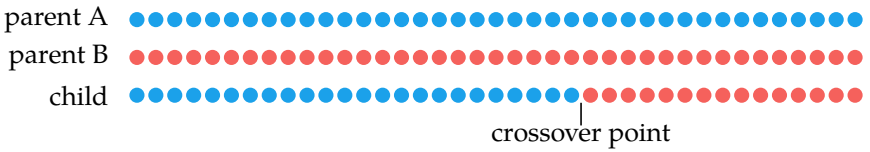


Figure 9.5. Single-point crossover.

- In *two-point crossover* (figure 9.6), we use two random crossover points.

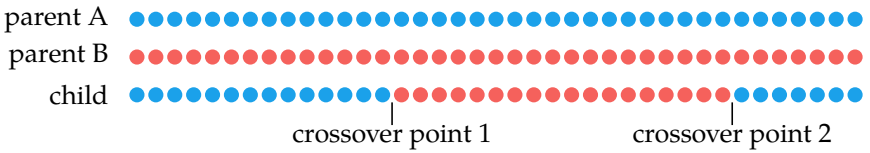


Figure 9.6. Two-point crossover.

- In *uniform crossover* (figure 9.7), each gene is independently selected with probability p from the second parent and probability $1 - p$ from the first parent. Typically, $p = 0.5$.

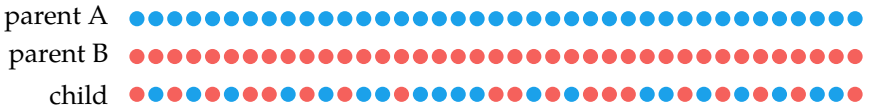


Figure 9.7. Uniform crossover.

- In *interpolation crossover*, values are linearly interpolated between the values of the parents $\mathbf{x}_a$ and $\mathbf{x}_b$:

$$\mathbf{x} \leftarrow (1 - \lambda)\mathbf{x}_a + \lambda\mathbf{x}_b \tag{9.1}$$

where λ is a scalar parameter typically set to 0.5.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
abstract type CrossoverMethod end

struct SinglePointCrossover <: CrossoverMethod end

function crossover(::SinglePointCrossover, a, b)
    i = rand(eachindex(a))
    return [a[1:i]; b[i+1:end]]
end

struct TwoPointCrossover <: CrossoverMethod end

function crossover(::TwoPointCrossover, a, b)
    n = length(a)
    i, j = rand(1:n, 2)
    if i > j
        (i,j) = (j,i)
    end
    return [a[1:i]; b[i+1:j]; a[j+1:n]]
end

struct UniformCrossover <: CrossoverMethod
    p # crossover probability
end

function crossover(U::UniformCrossover, a, b)
    return [rand() > U.p ? u : v for (u,v) in zip(a,b)]
end

struct InterpolationCrossover <: CrossoverMethod
    λ # interpolant
end

crossover(C::InterpolationCrossover, a, b) = (1-C.λ)*a + C.λ*b
```

Algorithm 9.4. Several crossover methods for genetic algorithms. Calling crossover with a `CrossoverMethod` and two parents `a` and `b` will produce a child chromosome that contains a mixture of the parents' genetic codes.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

9.2.3 *Mutation*

If new chromosomes were produced only through crossover, many traits that were not present in the initial random population could never occur, and the most-fit genes could saturate the population. Mutation allows new traits to spontaneously appear, allowing the genetic algorithm to explore more of the state space. Child chromosomes undergo mutation after crossover.

Each gene in the chromosome typically has a small probability λ of being changed. For a chromosome with m genes, this *mutation rate* is typically set to $1/n$, yielding an average of one mutation per child chromosome. Mutation can be implemented in various ways, but a common way is to add noise selected from a distribution such as the zero-mean Gaussian distribution. Algorithm 9.5 provides implementations.

```
abstract type MutationMethod end

struct DistributionMutation <: MutationMethod
    λ # mutation rate
    D # mutation distribution
end

function mutate(M::DistributionMutation, child)
    return [rand() < M.λ ? v + rand(M.D) : v for v in child]
end

GaussianMutation(σ) = DistributionMutation(1.0, Normal(0,σ))
```

Algorithm 9.5. A mutation method for genetic algorithms that iterates through each gene in a chromosome and, with probability `λ`, adds a random value drawn from the distribution `D`. Gaussian mutation is a special case of this where all genes are perturbed by a zero-mean Gaussian distribution with standard deviation `σ`. Other mutation methods might include bit flips if the chromosome is binary-valued.

Figure 9.8 illustrates several generations of a genetic algorithm. Example 9.1 shows how to combine selection, crossover, and mutation strategies discussed in this section.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

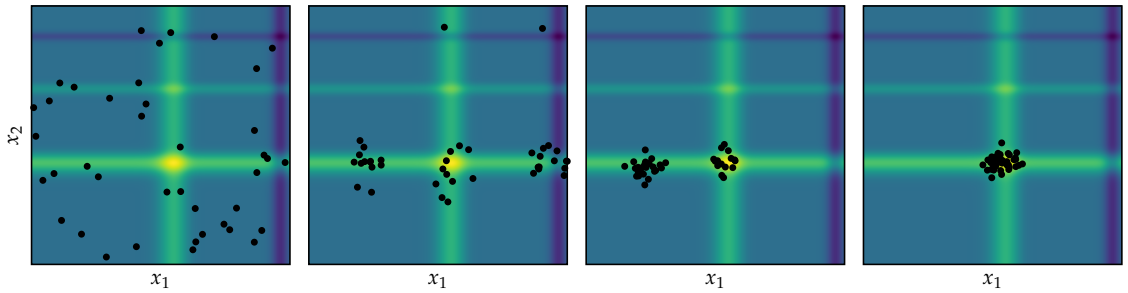


Figure 9.8. A genetic algorithm with truncation selection, single-point crossover, and Gaussian mutation with $\sigma = 0.1$ on the Michalewicz function (appendix B.5).

We demonstrate using genetic algorithms to optimize $f(\mathbf{x}) = \|\mathbf{x}\|$.

```
import Random: seed!
import LinearAlgebra: norm
seed!(0) # set random seed for reproducible results
f = x->norm(x)
m = 100 # population size
k_max = 10 # number of iterations
function rand_population_uniform(m, a, b)
    d = length(a)
    return [a+rand(d).*(b-a) for i in 1:m]
end
population = rand_population_uniform(m, [-3,-3], [3,3])
M = GeneticAlgorithm(
    TruncationSelection(10), # select top 10
    SinglePointCrossover(),
    GaussianMutation(0.5)) # perturb with 0.5 standard deviation
population = population_method(M, f, population, k_max)
x = argmin(f, population)
@show x
```

```
x = [-0.030867249413217987, 0.08635826032152316]
```

Example 9.1. Demonstration of using a genetic algorithm for optimizing a simple function.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

9.3 Differential Evolution

Differential evolution (algorithm 9.6) attempts to improve each individual in the population using crossover with an interim design formed from the recombination of other individuals in the population.[7] It is parameterized by a differential weight w and crossover probability p.

For each individual $\mathbf{x}$:

1. Choose three random distinct individuals $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$.
2. Construct an interim design $\mathbf{z} = \mathbf{a} + w \cdot (\mathbf{b} - \mathbf{c})$ as shown in figure 9.9.[8]
3. Construct the candidate individual $\mathbf{x}'$ through uniform crossover with $\mathbf{x}$ and $\mathbf{z}$ with probability p,[9] where

$$x_i' = \begin{cases} z_i & \text{with probability } p \\ x_i & \text{otherwise} \end{cases} \tag{9.2}$$

4. Insert the better design between $\mathbf{x}$ and $\mathbf{x}'$ into the next generation.

The interim design is a base individual $\mathbf{a}$ with an added perturbation $w \cdot (\mathbf{b} - \mathbf{c})$. The perturbation is necessary for exploration, and constructing it from the difference of two individuals allows the perturbations to naturally decrease as the population becomes more concentrated. Applying the perturbation to a base individual $\mathbf{a}$ rather than $\mathbf{x}$ itself allows information about good designs to be shared across individuals in the population. The crossover probability parameter controls how quickly the population will concentrate. Larger crossover probabilities will lead to faster concentration. The algorithm is demonstrated in figure 9.10.

[7] S. Das and P. N. Suganthan, "Differential Evolution: A Survey of the State-of-the-Art," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011.

[8] Typically, w is between 0 and 2.

[9] Studies on test function suites found that differential evolution performed best with either $p \approx 1$ or with $p \approx 0$ and one guaranteed mutation. This bifurcation was later attributed to whether the objective function was decomposable, with low mutation rates working better when the test function can be written as the sum of functions over disjoint subsets of the design. For such functions, steps along a single coordinate are more likely to lead to improvements, and those improvements can reliably be shared through crossover with other designs. K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*. Springer, 2006.

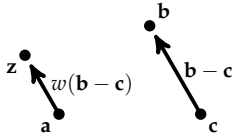


Figure 9.9. Differential evolution takes three individuals $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ and combines them to form the interim design $\mathbf{z}$.

9.4 Particle Swarm Optimization

Particle swarm optimization introduces momentum to accelerate convergence toward minima.[10] Each individual, or *particle*, in the population keeps track of its current position, velocity, and the best position it has seen so far. Momentum allows an individual to accumulate speed in a favorable direction, independent of local perturbations.

[10] J. Kennedy, R. C. Eberhart, and Y. Shi, *Swarm Intelligence*. Morgan Kaufmann, 2001.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct DifferentialEvolution <: PopulationMethod
    p # crossover probability
    w # differential weight
end

init!(M::DifferentialEvolution, f, designs) = designs

function step!(M::DifferentialEvolution, f, population)
    p, w = M.p, M.w
    n, m = length(population[1]), length(population)
    for x in population
        a, b, c = sample(population, 3, replace=false)
        z = a + w*(b-c)
        x′ = crossover(UniformCrossover(p), x, z)
        if f(x′) < f(x)
            x .= x′
        end
    end
    return population
end
```

Algorithm 9.6. Differential evolution, which updates populations through recombination with three other population members. The algorithm requires a crossover probability `p` and a differential weight `w`. This implementation makes two simplifications to the formulation typically presented in the literature. First, our implementation does not enforce that **a**, **b**, and **c** be distinct from **x**. In practice, they will be for large populations, but even if they are not, it is not particularly detrimental. Second, the original formulation ensures that uniform crossover has at least one mutation. When the mutation rate is very low, it is necessary to ensure that at least one mutation occurs in order for the candidate individual to differ.

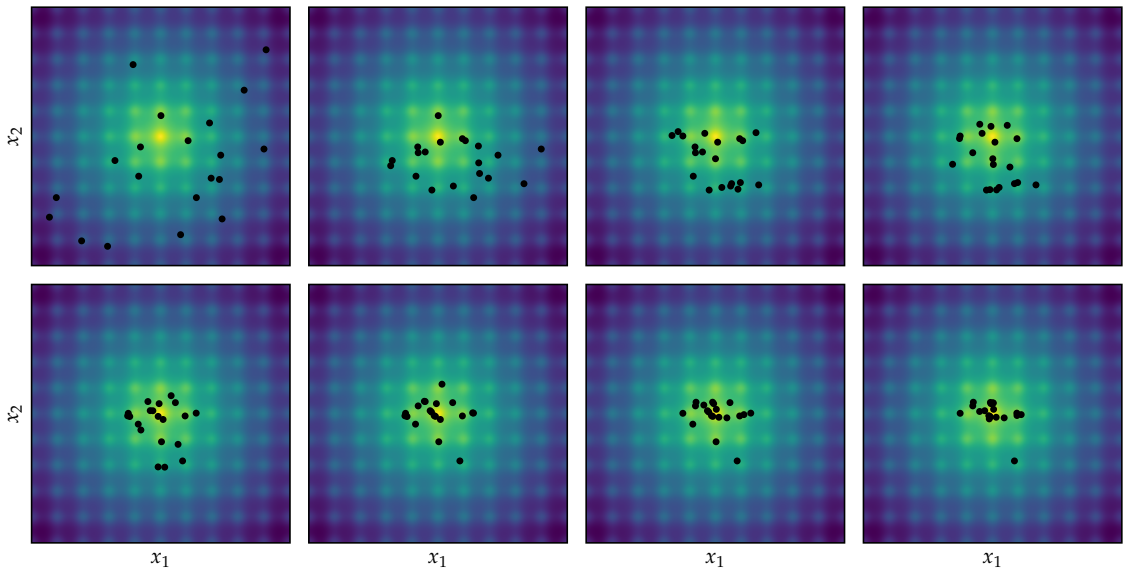


Figure 9.10. Differential evolution with $p = 0.9$ and $w = 0.4$ on Ackley's function (appendix B.1).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

At each iteration, each individual is accelerated toward both the best position it has seen and the best position found thus far by any individual. The components of the acceleration vector are weighted by random values. The update equations are:

$$\mathbf{v}^{(i)} \leftarrow w\mathbf{v}^{(i)} + c_1\mathbf{r}_1 \odot \left(\mathbf{x}_{\text{best}}^{(i)} - \mathbf{x}^{(i)}\right) + c_2\mathbf{r}_2 \odot \left(\mathbf{x}_{\text{best}} - \mathbf{x}^{(i)}\right) \tag{9.3}$$

$$\mathbf{x}^{(i)} \leftarrow \mathbf{x}^{(i)} + \mathbf{v}^{(i)} \tag{9.4}$$

where $\mathbf{x}_{\text{best}}$ is the best location found so far over all particles; $\mathbf{x}_{\text{best}}^{(i)}$ is the best location found by the ith particle; w, c_1, and c_2 are parameters; and $\mathbf{r}_1$ and $\mathbf{r}_2$ are random vectors with components drawn from the uniform distribution $\mathcal{U}(0, 1)$.[11] Algorithm 9.7 provides an implementation. Figure 9.11 shows several iterations of the algorithm.

[11] A common strategy is to allow the inertia w to decay over time.

9.5 Firefly Algorithm

The *firefly algorithm* (algorithm 9.8) was inspired by the manner in which fireflies flash their lights to attract mates of the same species.[12] In the firefly algorithm, each individual in the population is a firefly and can flash to attract other fireflies. At each iteration, all fireflies are moved toward all more attractive fireflies. A firefly $\mathbf{a}$ is moved toward a firefly $\mathbf{b}$ with greater attraction according to

$$\mathbf{a} \leftarrow \mathbf{a} + \beta I(\|\mathbf{b} - \mathbf{a}\|)(\mathbf{b} - \mathbf{a}) + \alpha\boldsymbol{\epsilon} \tag{9.5}$$

where I is the attraction intensity and β is a scaling parameter. A random walk component is included as well, where $\boldsymbol{\epsilon}$ is drawn from a zero-mean, unit covariance multivariate Gaussian, and α scales the step size. The resulting update is a random walk biased toward brighter fireflies.[13]

[12] X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms*, 2nd ed. Luniver Press, 2010. Interestingly, some females imitate the patterns of other species to attract the males of the other species, which they then eat.

[13] Yang recommends $\beta = 1$ and $\alpha \in [0, 1]$. If $\beta = 0$, the behavior is a random walk.

The intensity I decreases as the distance r between the two fireflies increases and is defined to be 1 when $r = 0$. One approach is to model the intensity with a Gaussian brightness drop-off:

$$I(r) = e^{-\gamma r^2} \tag{9.6}$$

where $\gamma > 0$ is a parameter that controls the rate of decay.

A firefly's attraction is proportional to its performance. Attraction affects only whether one fly is attracted to another fly, whereas intensity affects how much the less attractive fly moves toward the more attractive fly. Figure 9.12 shows a few iterations of the algorithm.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct Particle
    x      # position
    v      # velocity
    x_best # best design thus far
end

mutable struct ParticleSwarm <: PopulationMethod
    w  # inertia
    c1 # first momentum coefficient
    c2 # second momentum coefficient
    V  # initial particle velocity distribution
    best # best overall design thus far, and its value
end

function init!(M::ParticleSwarm, f, designs)
    population = [Particle(x,rand(M.V),copy(x)) for x in designs]
    best = (x=copy(population[1].x), y=Inf)
    for P in population
        y = f(P.x)
        if y < best.y; best = (x=P.x, y=y); end
    end
    M.best = best
    return population
end

function step!(M::ParticleSwarm, f, population)
    w, c1, c2, best = M.w, M.c1, M.c2, M.best
    n = length(best.x)
    for P in population
        r1, r2 = rand(n), rand(n)
        P.v = w*P.v + c1*r1.*(P.x_best - P.x) +
                      c2*r2.*(best.x - P.x)
        P.x += P.v
        y = f(P.x)
        if y < best.y; best = (x=copy(P.x), y=y); end
        if y < f(P.x_best); P.x_best .= P.x; end
    end
    M.best = best
    return population
end
```

Algorithm 9.7. Particle swarm optimization, which incorporates momentum into population updates. It requires an inertia `w`, momentum coefficients `c1` and `c2`, and an initial velocity distribution `V`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

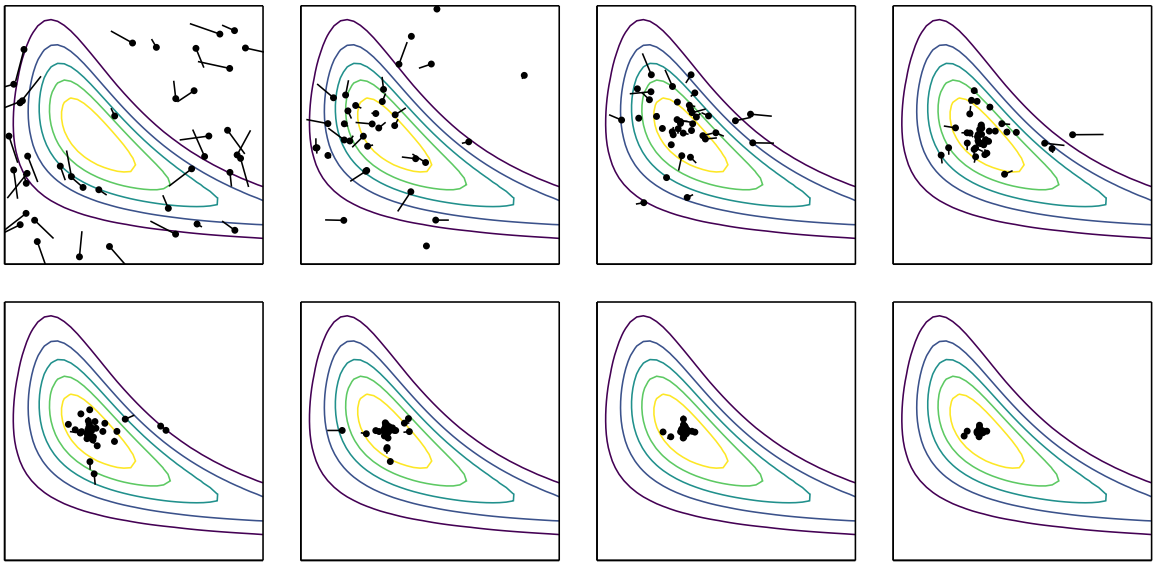


Figure 9.11. The particle swarm method with $w = 0.1$, $c_1 = 0.25$, and $c_2 = 2$ on Wheeler's Ridge (appendix B.7).

```
struct Firefly <: PopulationMethod
    α          # walk step size
    β          # source intensity
    brightness # intensity function
end

init!(M::Firefly, f, designs) = designs

function step!(M::Firefly, f, population)
    α, β, brightness = M.α, M.β, M.brightness
    m = length(population[1])
    N = MvNormal(I(m))
    for a in population, b in population
        if f(b) < f(a)
            r = norm(b-a)
            a .+= β*brightness(r)*(b-a) + α*rand(N)
        end
    end
    return population
end
```

Algorithm 9.8. Firefly search, where individuals are attracted to one another according to their luminance. The method requires a random walk step size `α`, a source intensity `β`, and an intensity function `brightness` such as `r->exp(-r^2)`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

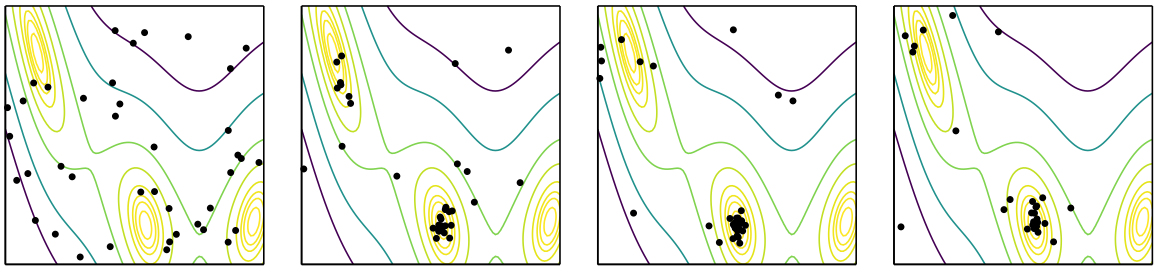


Figure 9.12. Firefly search with $\alpha = 0.5$, $\beta = 1$, and $\gamma = 0.1$ applied to the Branin function (appendix B.3).

9.6 Cuckoo Search

Cuckoo search (algorithm 9.9) is another nature-inspired algorithm named after the cuckoo bird, which engages in a form of brood parasitism.[14] Cuckoos lay their eggs in the nests of other birds, often birds of other species. When this occurs, the host bird may detect the invasive egg and then destroy it or establish a new nest somewhere else. However, there is also a chance that the egg is accepted and raised by the host bird.[15]

In cuckoo search, each nest represents a design point. New design points can be produced using *Lévy flights* from nests, which are random walks with step-lengths from a heavy-tailed distribution. A new design point can replace a nest if it has a better objective function value, which is analogous to cuckoo eggs replacing the eggs of birds of other species.

The core rules are:

1. A cuckoo will lay an egg in a randomly chosen nest.
2. The best nests with the best eggs will survive to the next generation.
3. Cuckoo eggs have a chance of being discovered by the host bird, in which case the eggs are destroyed.

Cuckoo search relies on random flights to establish new nest locations. These flights start from an existing nest and then move randomly to a new location. While we might be tempted to use a uniform or Gaussian distribution for the walk, these restrict the search to a relatively concentrated region. Instead, cuckoo search uses a Cauchy distribution, which has a heavier tail. In addition, the Cauchy distribution has been shown to be more representative of the movements of other animals in the wild.[16] Figure 9.13 shows a few iterations of cuckoo search.

[14] X.-S. Yang and S. Deb, "Cuckoo Search via Lévy Flights," in *World Congress on Nature & Biologically Inspired Computing* (*NaBIC*), 2009.

[15] Interestingly, an instinct of newly hatched cuckoos is to knock other eggs or hatchlings (those belonging to the host bird) out of the nest.

[16] For example, a certain species of fruit fly explores its surroundings using Cauchy-like steps separated by 90° turns. A. M. Reynolds and M. A. Frye, "Free-Flight Odor Tracking in Drosophila is Consistent with an Optimal Intermittent Scale-Free Search," *PLoS ONE*, vol. 2, no. 4, e354, 2007.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct CuckooSearch <: PopulationMethod
    p_s # search fraction
    p_a # nest abandonment fraction
    C   # flight distribution
end

function init!(M::CuckooSearch, f, designs)
    return [(x=x, y=f(x)) for x in designs]
end

function step!(M::CuckooSearch, f, population)
    p_s, p_a, C = M.p_s, M.p_a, M.C
    m, n = length(population), length(population[1].x)
    m_search = round(Int, m*p_s)
    m_abandon = round(Int, m*p_a)
    for i in 1:m_search
        j, k = rand(1:m), rand(1:m)
        x = population[j].x + rand(C,n)
        y = f(x)
        if y < population[k].y
            population[k] = (x=x, y=y)
        end
    end
    p = sortperm(population, by=nest->nest.y, rev=true)
    for i in 1:m_abandon
        j = rand(1:m-m_abandon)+m_abandon
        x' = population[p[j]].x + rand(C,n)
        population[p[i]] = (x=x', y=f(x'))
    end
    return population
end
```

Algorithm 9.9. Cuckoo search, which takes an objective function `f`, an initial set of nests `population`, search fraction `p_s`, fraction of nests to abandon `p_a`, and flight distribution `C`. The flight distribution is typically a centered Cauchy distribution.

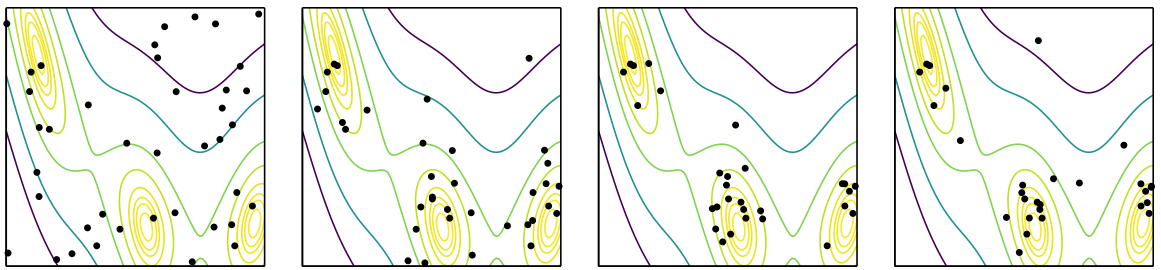


Figure 9.13. Cuckoo search on the Branin function (appendix B.3).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Other nature-inspired algorithms include the artificial bee colony, the gray wolf optimizer, the bat algorithm, glowworm swarm optimization, intelligent water drops, and harmony search.[17] There has been some criticism of the proliferation of methods that make analogies to nature without fundamentally contributing novel methods and understanding.[18]

[17] D. Simon, *Evolutionary Optimization Algorithms*. Wiley, 2013.

[18] This viewpoint is expressed by K. Sörensen, "Metaheuristics—the Metaphor Exposed," *International Transactions in Operational Research*, vol. 22, no. 1, pp. 3–18, 2015.

9.7 Hybrid Methods

Many population methods perform well in global search, being able to avoid local minima and finding the best regions of the design space. Unfortunately, these methods do not perform as well in local search in comparison to descent methods. Several *hybrid methods*[19] have been developed to extend population methods with descent-based features to improve their performance in local search. There are two general approaches to combining population methods with local search techniques:[20]

[19] In the literature, these kinds of techniques are also referred to as *memetic algorithms* or *genetic local search*.

[20] K. W. C. Ku and M.-W. Mak, "Exploring the Effects of Lamarckian and Baldwinian Learning in Evolving Recurrent Neural Networks," in *IEEE Congress on Evolutionary Computation (CEC)*, 1997.

- In *Lamarckian learning*, the population method is extended with a local search method that locally improves each individual. The original individual and its objective function value are replaced by the individual's optimized counterpart and its objective function value.
- In *Baldwinian learning*, the same local search method is applied to each individual, but the results are used only to update the individual's objective function value. Individuals are not replaced but are merely associated with optimized objective function values, which are not the same as their actual objective function values. Baldwinian learning can help prevent premature convergence.

The difference between these approaches is illustrated in example 9.2.

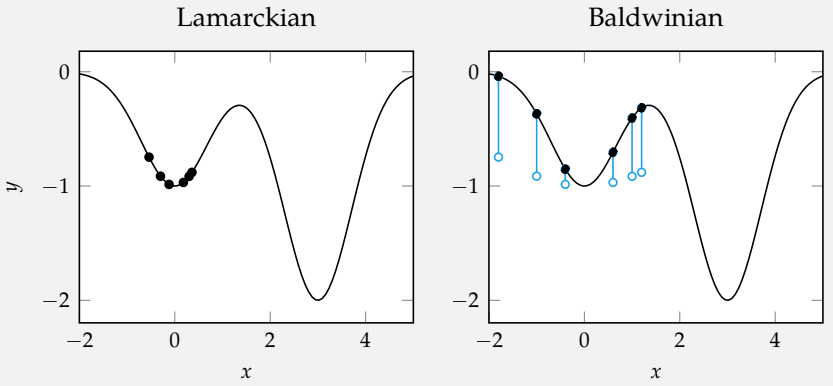
9.8 Summary

- Population methods use a collection of individuals in the design space to guide progression toward an optimum.
- Genetic algorithms leverage selection, crossover, and mutations to produce better subsequent generations.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 9.2. A comparison of the Lamarckian and Baldwinian hybrid methods.

Consider optimizing $f(x) = -e^{-x^2} - 2e^{-(x-3)^2}$ using a population of individuals initialized near $x = 0$.



A Lamarckian local search update applied to this population would move the individuals toward the local minimum, reducing the chance that future individuals escape and find the global optimum near $x = 3$. A Baldwinian approach would perform the same local search but leave the original designs unchanged. During selection, each design would be evaluated using the objective value obtained after local search.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- Differential evolution, particle swarm optimization, the firefly algorithm, and cuckoo search include rules and mechanisms for attracting design points to the best individuals in the population while maintaining suitable state space exploration.
- Population methods can be extended with local search approaches to improve convergence.

9.9 Exercises

Exercise 9.1. What is the motivation behind the selection operation in genetic algorithms?

Solution: To bias survival to the fittest by biasing the selection toward the individuals with better objective function values.

Exercise 9.2. Why does mutation play such a fundamental role in genetic algorithms? How would we choose the mutation rate if we suspect there is a better optimal solution?

Solution: Mutation drives exploration using randomness. It is therefore essential in order to avoid local minima. If we suspect there is a better solution, we would need to increase the mutation rate and let the algorithm have time to discover it.

Exercise 9.3. If we observe that particle swarm optimization results in fast convergence to a nonglobal minimum, how might we change the parameters of the algorithm?

Solution: Increase the population size or the coefficient that biases the search toward individual minima.

Exercise 9.4. Differential evolution combines uniform crossover with a recombination of three other individuals in the population. Compare the following five variants of differential evolution on Ackley's function (appendix B.1):

- Uniform crossover with one chosen individual ($p = 0.5$, $w = 0.0$).
- Perturbations applied to the design ($p = 0.5$, $w = 0.5$, $\mathbf{a} = \mathbf{x}$).
- Differential evolution with ($p = 0.5$, $w = 0.5$).
- A high crossover probability ($p = 0.9$, $w = 0.5$).
- Guaranteed crossover ($p = 1.0$, $w = 0.5$).

How do the various components of differential evolution contribute to its performance?

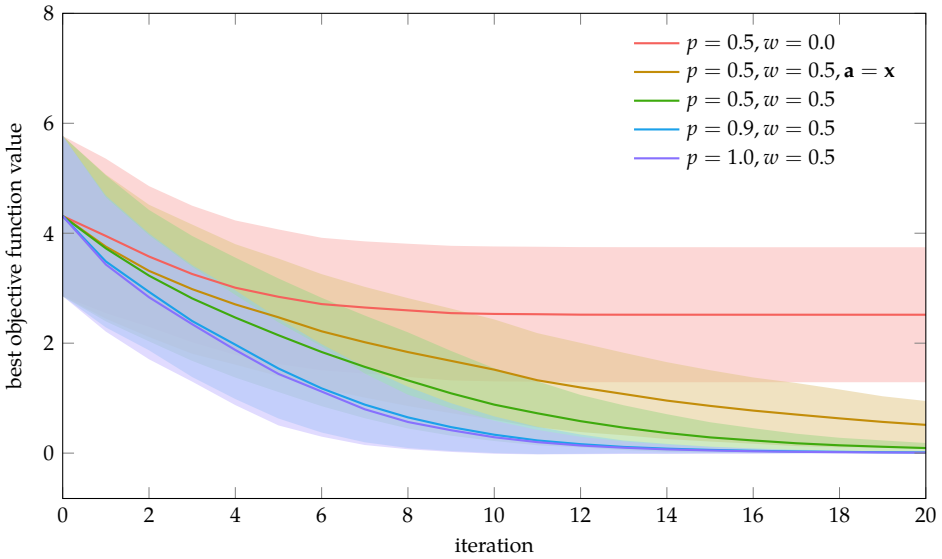
Solution: Below we show the distributions over the best value found so far over 1,000 trials each with populations of size 20 for the five algorithm variations.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Removing the perturbation vector ($w = 0$) does the worst and tends not to converge to an optimum. The algorithm can only explore designs with components that already exist in the population; it cannot produce new design components.

Perturbations applied directly to the design ($\mathbf{a} = \mathbf{x}$) are the next worst. It forces all updates to be local perturbations, which makes it difficult for the algorithm to escape local minima.

The method that does the best is the one with guaranteed crossover ($p = 1.0$), which always replaces all components of $\mathbf{x}$ with the interim design. Having a large but non-guaranteed crossover probability ($p = 0.9$) does nearly as well.



This exercise demonstrates that differential evolution's construction of an interim design through $\mathbf{a} + w \cdot (\mathbf{b} - \mathbf{c})$ relies on $w \cdot (\mathbf{b} - \mathbf{c})$ to supply a perturbation that can be used to explore the space, and relies on $\mathbf{a}$ to allow the method to abandon designs in poor local minima by jumping close to $\mathbf{a}$. This particular experiment found less need for uniform crossover, but our design space only had two dimensions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

10 Constraints

Previous chapters have focused on unconstrained problems where the domain of each design variable is the space of real numbers. Many problems are constrained, which forces design points to satisfy certain conditions. This chapter presents a variety of approaches for transforming problems with constraints into problems without constraints, thereby permitting the use of the optimization algorithms we have already discussed. Analytical methods are also discussed, including the necessary conditions for optimality with constraints.

10.1 Constrained Optimization

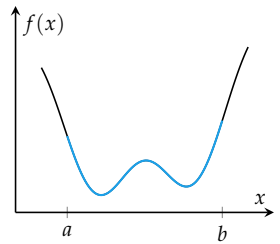
Recall the core optimization problem from equation (1.1):

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x} \in \mathcal{X} \end{aligned} \tag{10.1}$$

In the previous chapters, the feasible set $\mathcal{X}$ was assumed to be $\mathbb{R}^n$. In the constrained problems in this chapter, the feasible set is some subset of $\mathbb{R}^n$.

Some constraints are simply upper or lower bounds on the design variables, as we have seen in bracketed line search, in which x must lie between a and b. A bracketing constraint $x \in [a, b]$ can be replaced by two inequality constraints: $a \leq x$ and $x \leq b$ as shown in figure 10.1. In multivariate problems, bracketing the input variables forces them to lie within a hyperrectangle as shown in figure 10.2.

Constraints arise naturally when formulating real problems. A hedge fund manager cannot sell more stock than they have, an airplane cannot have wings with zero thickness, and the number of hours you spend per week on your homework cannot exceed 168. We include constraints in such problems to prevent the optimization algorithm from suggesting an infeasible solution.



$$\begin{aligned} &\underset{x}{\text{minimize}} && f(x) \\ &\text{subject to} && x \in [a, b] \end{aligned}$$

Figure 10.1. A simple optimization problem constrained by upper and lower bounds.

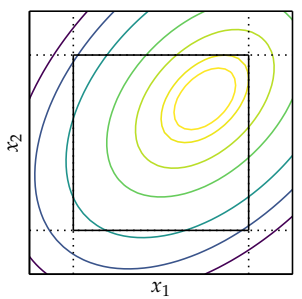


Figure 10.2. Bracketing constraints of the form $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$ force the solution to lie within a hyperrectangle. Such constraints are often referred to as *box constraints*.

Applying constraints to a problem can affect the solution, but this need not be the case as shown in figure 10.3.

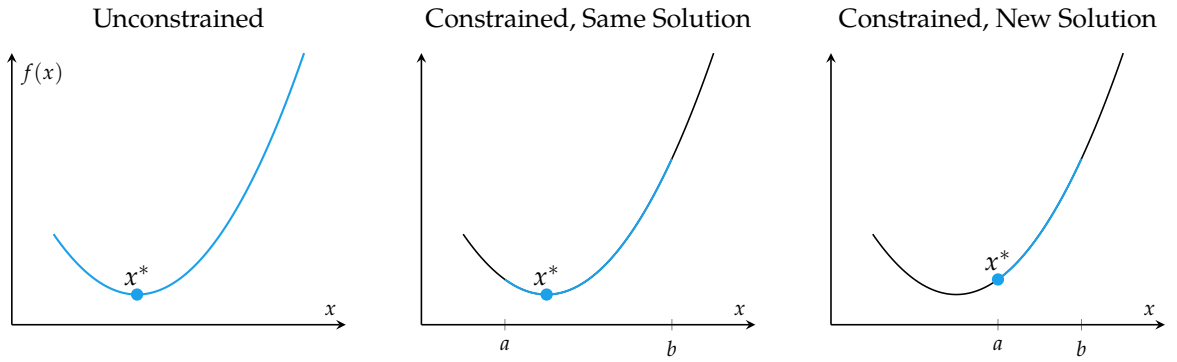


Figure 10.3. Constraints can change the solution to a problem but do not have to.

10.2 Constraint Types

This chapter focuses on representing $\mathcal{X}$ using two types of constraints:

1. equality constraints, $h(\mathbf{x}) = 0$
2. inequality constraints, $g(\mathbf{x}) \leq 0$

We have g representing a less-than inequality constraint. Greater-than inequality constraints can be translated into less-than inequality constraints by introducing a negative sign.

Any optimization problem can be rewritten using these constraints:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & h_i(\mathbf{x}) = 0 \qquad \text{for } i \text{ in } 1:\ell \\ & g_j(\mathbf{x}) \leq 0 \qquad \text{for } j \text{ in } 1:m \end{aligned} \tag{10.2}$$

Of course, we can convert an arbitrary set-membership constraint $x \in \mathcal{X}$ into an equality constraint:

$$h(\mathbf{x}) = (\mathbf{x} \notin \mathcal{X}) \tag{10.3}$$

where Boolean expressions in parentheses evaluate to 0 if false and 1 if true.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We often use equality and inequality functions ($h(\mathbf{x}) = 0$, $g(\mathbf{x}) \leq 0$) to define constraints rather than set membership because the functions can provide information about how far a given point is from being feasible. This information helps drive solution methods toward feasibility.

Equality constraints are sometimes decomposed into two inequality constraints:

$$h(\mathbf{x}) = 0 \quad \iff \quad \begin{cases} h(\mathbf{x}) \leq 0 \\ h(\mathbf{x}) \geq 0 \end{cases} \tag{10.4}$$

However, sometimes we want to handle equality constraints separately, as we will discuss later in this chapter.

There are many other categories of constraints. For example, we may have constraints that require that some of the design variables be integers, as will be introduced in chapter 22. We may also have constraints that require that the design fall within a *cone*, as shown in figure 10.4. In some applications, our design space corresponds to the set of positive semidefinite matrices, which is a special type of cone that is the subject of *semidefinite programming*. Such problems can be solved using variations of the same techniques presented in this chapter, such as interior point methods.

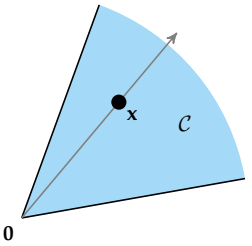


Figure 10.4. A set $\mathcal{C}$ is a cone if $\theta\mathbf{x} \in \mathcal{C}$ for every $\mathbf{x} \in \mathcal{C}$ and $\theta \geq 0$. Convex cones are of particular interest, which are cones where the line segment between any two points in the cone is entirely in the cone. A *cone constraint* is written $\mathbf{x} \preceq_{\mathcal{C}} \mathbf{y}$. Such a constraint is equivalent to $\mathbf{y} - \mathbf{x} \in \mathcal{C}$. These constraints are also called *generalized inequality constraints* because they generalize elementwise inequality constraints of the form $\mathbf{x} \leq \mathbf{y}$, which are equivalent to $\mathbf{y} - \mathbf{x} \in \mathbb{R}^n_+$ where $\mathbb{R}^n_+$ is the set of vectors with nonnegative components. S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

10.3 Transformations to Remove Constraints

In some cases, it may be possible to transform a problem so that constraints can be removed. For example, an interval constraint $a \leq x \leq b$ can be removed by passing x through a transform (figure 10.5):[1]

$$x = t_{a,b}(\hat{x}) = \frac{b+a}{2} + \frac{b-a}{2}\left(\frac{2\hat{x}}{1+\hat{x}^2}\right) \tag{10.5}$$

Example 10.1 demonstrates this process.

Some equality constraints can be used to solve for x_n given $x_1, \ldots, x_{n-1}$. In other words, if we know the first $n-1$ components of $\mathbf{x}$, we can use the constraint equation to obtain x_n. In such cases, the optimization problem can be reformulated over $x_1, \ldots, x_{n-1}$ instead, removing the constraint and removing one design variable. Example 10.2 demonstrates this process.

[1] This and other transforms are discussed by S. K. Park, "A Transformation Method for Constrained-Function Minimization," National Aeronautics and Space Administration, Technical Note TN D-7983, 1975.

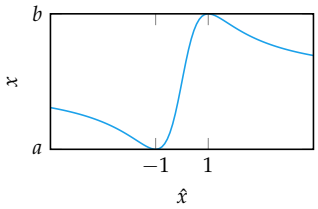


Figure 10.5. This transform ensures that x is between a and b.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 10.1. Removing bounds constraints using a transform on the input variable.

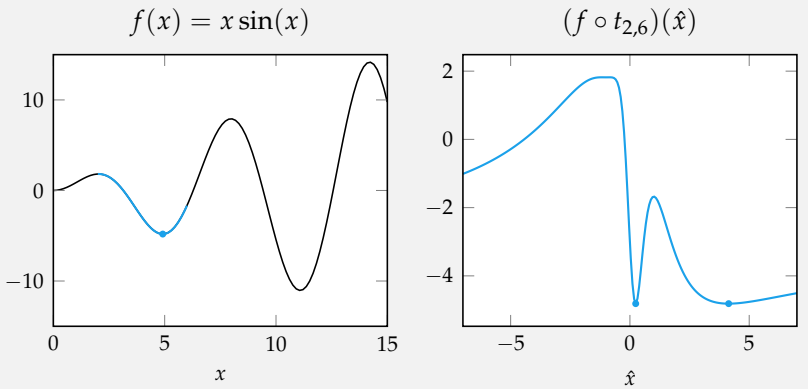
Consider the optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x \sin(x) \\ \text{subject to} \quad & 2 \leq x \leq 6 \end{aligned}$$

We can transform the problem to remove the constraints:

$$\begin{aligned} \underset{\hat{x}}{\text{minimize}} \quad & t_{2,6}(\hat{x}) \sin(t_{2,6}(\hat{x})) \\ \underset{\hat{x}}{\text{minimize}} \quad & \left(4 + 2\left(\frac{2\hat{x}}{1+\hat{x}^2}\right)\right) \sin\left(4 + 2\left(\frac{2\hat{x}}{1+\hat{x}^2}\right)\right) \end{aligned}$$

We can use the optimization method of our choice to solve the unconstrained problem. In doing so, we find two minima: $\hat{x} \approx 0.242$ and $\hat{x} \approx 4.139$, both of which have a function value of approximately -4.814.



The solution for the original problem is obtained by passing $\hat{x}$ through the transform. Both values of $\hat{x}$ produce $x = t_{2,6}(\hat{x}) \approx 4.914$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 10.2. Removing an equality constraint and design variable through a transformation.

Consider the constraint:

$$h(\mathbf{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n = 0$$

We can solve for x_n using the first $n-1$ variables:

$$x_n = \frac{1}{c_n}(-c_1x_1 - c_2x_2 - \cdots - c_{n-1}x_{n-1})$$

We can transform

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & h(\mathbf{x}) = 0 \end{aligned}$$

into

$$\underset{x_1,\ldots,x_{n-1}}{\text{minimize}} \quad f\left(\left[x_1, \ldots, x_{n-1}, \frac{1}{c_n}(-c_1x_1 - c_2x_2 - \cdots - c_{n-1}x_{n-1})\right]\right)$$

10.4 Removing Affine Equality Constraints

Many problems have affine equality constraints in the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ with $\mathbf{A} \in \mathbb{R}^{m\times n}$. If $\mathbf{A}$ has rank $m = n$, then $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution. When $\mathbf{A}$ has rank $m < n$, we can reformulate the problem with fewer design variables and eliminate the affine constraint. To do this, we begin by computing an LQ decomposition[2] of $\mathbf{A}$:

[2] See appendix C.8.3 for a review.

$$\mathbf{A} = \begin{bmatrix}\mathbf{L} & \mathbf{0}\end{bmatrix}\mathbf{Q} \tag{10.6}$$

where $\mathbf{L} \in \mathbb{R}^{m\times m}$ is lower-triangular and $\mathbf{Q} \in \mathbb{R}^{n\times n}$ is orthogonal. Our constraint becomes:

$$\begin{bmatrix}\mathbf{L} & \mathbf{0}\end{bmatrix}\mathbf{Q}\mathbf{x} = \mathbf{b} \tag{10.7}$$

We then apply a change of variables:

$$\mathbf{Q}\mathbf{x} = \mathbf{y} = \begin{bmatrix}\mathbf{y}_{1:m} \\ \mathbf{y}_{(m+1:n)}\end{bmatrix} \tag{10.8}$$

and find that our constraint is satisfied[3] when $\mathbf{y}^*_{1:m} = \mathbf{L}^{-1}\mathbf{b}$. The remaining variables in $\mathbf{y}_{(m+1:n)}$ can take on any values.

[3] We can efficiently solve for $\mathbf{y}^*_{1:m}$ using forward substitution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We then define a new problem with $n - m$ design variables by replacing all instances of $\mathbf{x}$ in our original problem with:[4]

$$\mathbf{x} = \mathbf{Q}^\top \mathbf{y} = \mathbf{Q}^\top \begin{bmatrix} \mathbf{y}^*_{1:m} \\ \mathbf{y}_{(m+1:n)} \end{bmatrix} \tag{10.9}$$

and remove the affine constraint. Once solved for $\mathbf{y}^*_{(m+1:n)}$, we use the same equation to recover $\mathbf{x}^*$.

[4] The inverse of an orthogonal matrix is equal to its transpose.

10.5 Projected Iterative Methods

We can use any of our iterative unconstrained methods discussed in the previous chapters on the unconstrained objective function and maintain feasibility by projecting the unconstrained iterate $\mathbf{x}_{\text{unc}}$ back to a feasible design that is closest:[5]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x} - \mathbf{x}_{\text{unc}}\| \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \end{aligned} \tag{10.10}$$

[5] This formulation uses the L_2-norm, but any distance metric can be used. If $\mathcal{X}$ is nonconvex, the projection may not be unique.

This projection can be straightforward, such as with box constraints where $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$. In this case, each component can be projected independently, setting x_i to the nearest bound if it lies outside the feasible interval.

In other cases, the projection may be more complex but still analytically solvable. For example, if the feasible set is defined by a single linear inequality constraint $\mathbf{a}^\top \mathbf{x} \leq b$, the projection to the feasible set is simply

$$\mathbf{x}_{\text{unc}} - \frac{\max\left(0, \mathbf{a}^\top \mathbf{x}_{\text{unc}} - b\right)}{\mathbf{a}^\top \mathbf{a}} \mathbf{a} \tag{10.11}$$

This projection can be done in time linear in the number of design variables.

If there are multiple linear equality and inequality constraints, as shown in figure 10.6, the projection involves solving:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x} - \mathbf{x}_{\text{unc}}\| \\ \text{subject to} \quad & \mathbf{C}_{\text{LE}}\mathbf{x} \leq \mathbf{d}_{\text{LE}} \\ & \mathbf{C}_{\text{EQ}}\mathbf{x} = \mathbf{d}_{\text{EQ}} \end{aligned} \tag{10.12}$$

Squaring the objective yields a quadratic objective with linear constraints, making it a type of quadratic program (chapter 13) that can be solved in polynomial time. In general, when $\mathcal{X}$ is nonconvex, finding the projection of $\mathbf{x}_{\text{unc}}$ to a feasible set $\mathcal{X}$ can be intractable.

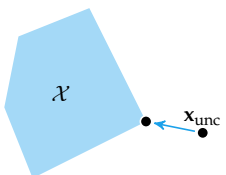


Figure 10.6. Projecting an unconstrained iterate to a feasible set.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Algorithm 10.1 implements *projected descent*, where at each descent iteration, the unconstrained iterate is projected back to the feasible set. While this implementation uses a descent method as the unconstrained optimizer, this idea can be applied broadly to any iterative unconstrained optimization method, including the population methods discussed in the previous chapter.

```
function projected_descent(M::DescentMethod, f, ∇f, x, X, k_max)
    x = project_to_feasible_set(X, x)
    init!(M, f, ∇f, x)
    for k in 1:k_max
        x = step!(M, f, ∇f, x)
        x = project_to_feasible_set(X, x)
    end
    return x
end
```

Algorithm 10.1. Projected descent for minimizing `f` starting at design `x` using information from the gradient `∇f` based on algorithm 4.1. Projection to a representation of the feasible set `X` is done by a `project_to_feasible_set` call, which must be tailored to the type of feasible set.

10.6 Lagrange Multipliers

The method of *Lagrange multipliers* is used to optimize a function subject to equality constraints.[6] Consider an optimization problem with a single equality constraint:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & h(\mathbf{x}) = 0 \end{aligned} \tag{10.13}$$

where f and h have continuous partial derivatives. Example 10.3 discusses such a problem.

If a point $\mathbf{x}^*$ minimizes f along the contour $h(\mathbf{x}) = 0$, then the directional derivative of f at $\mathbf{x}^*$ along $h(\mathbf{x}) = 0$ must be zero. That is, small shifts of $\mathbf{x}^*$ along $h(\mathbf{x}) = 0$ cannot result in an improvement. The method of Lagrange multipliers is used to compute where a contour line of f is aligned with the contour line of $h(\mathbf{x}) = 0$.

Since the gradient of a function at a point is perpendicular to the contour line of that function through that point, we know the gradient of h will be perpendicular to the contour line $h(\mathbf{x}) = 0$. Hence, we need to find where the gradient of f and the gradient of h are aligned.

Two vectors are aligned if one is a scalar multiple of the other, so we seek the best $\mathbf{x}$ such that the constraint

$$h(\mathbf{x}) = 0 \tag{10.14}$$

[6] Named for the Italian mathematician, physicist, and astronomer Joseph-Louis Lagrange (1736–1813).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 10.3. A motivating example of the method of Lagrange multipliers.

Consider the minimization problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & -\exp\left(-\left(x_1 x_2 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right) \\ \text{subject to} \quad & x_1 - x_2^2 = 0 \end{aligned}$$

We substitute the constraint $x_1 = x_2^2$ into the objective function to obtain an unconstrained objective:

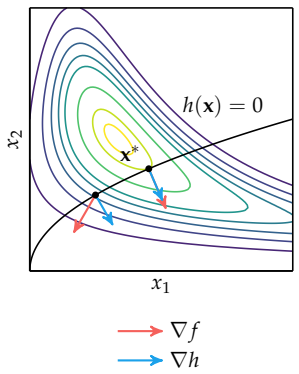
$$f_{\text{unc}} = -\exp\left(-\left(x_2^3 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right)$$

whose derivative is:

$$\frac{\partial}{\partial x_2} f_{\text{unc}} = 6\exp\left(-\left(x_2^3 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right)\left(x_2^5 - \frac{3}{2}x_2^2 + \frac{1}{3}x_2 - \frac{1}{2}\right)$$

Setting the derivative to zero and solving for x_2 yields $x_2 \approx 1.165$. The solution to the original optimization problem is thus $\mathbf{x}^* \approx [1.358, 1.165]$. The optimum lies where the contour line of f is aligned with h.

The contour lines of f represent curves along which the value of f remains constant. If a contour line of f is tangent to the curve $h(x) = 0$, then the directional derivative of f at that point, along the curve, must be zero.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

is satisfied and the gradients are aligned with

$$\nabla f(\mathbf{x}) = \lambda \nabla h(\mathbf{x}) \tag{10.15}$$

for some *Lagrange multiplier* λ. We need the scalar λ because the magnitudes of the gradients may not be the same, and they may have opposite signs. When ∇f is zero, the Lagrange multiplier λ equals zero, irrespective of ∇h.

When solving constrained optimization problems, we often use what is called the *Lagrangian*. It is a function of the design variables and the multiplier:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda h(\mathbf{x}) \tag{10.16}$$

Solving $\nabla \mathcal{L}(\mathbf{x}, \lambda) = \mathbf{0}$ solves equations (10.14) and (10.15). Setting $\nabla_{\mathbf{x}} \mathcal{L} = \mathbf{0}$ gives us the condition $\nabla f = \lambda \nabla h$, and $\nabla_\lambda \mathcal{L} = 0$ gives us $h(\mathbf{x}) = 0$. Any solution is considered a *critical point*. Critical points can be local minima, global minima, or saddle points.[7] Example 10.4 demonstrates this approach.

[7] The method of Lagrange multipliers gives us a first-order necessary condition to test for optimality. We will extend this method to include inequalities.

Example 10.4. Using the method of Lagrange multipliers to solve the problem in example 10.3.

We can use the method of Lagrange multipliers to solve the problem in example 10.3. We form the Lagrangian

$$\mathcal{L}(x_1, x_2, \lambda) = -\exp\left(-\left(x_1 x_2 - \frac{3}{2}\right)^2 - \left(x_2 - \frac{3}{2}\right)^2\right) - \lambda(x_1 - x_2^2)$$

and compute the gradient

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x_1} &= 2x_2 f(\mathbf{x})\left(\frac{3}{2} - x_1 x_2\right) - \lambda \\
\frac{\partial \mathcal{L}}{\partial x_2} &= 2\lambda x_2 + f(\mathbf{x})\left(-2x_1(x_1 x_2 - \frac{3}{2}) - 2(x_2 - \frac{3}{2})\right) \\
\frac{\partial \mathcal{L}}{\partial \lambda} &= x_2^2 - x_1
\end{aligned}$$

Setting these derivatives to zero and solving yields $x_1 \approx 1.358$, $x_2 \approx 1.165$, and $\lambda \approx 0.170$.

The method of Lagrange multipliers can be extended to multiple equality constraints. Here, rather than a single directional derivative being zero, the directional derivative of f along every constraint contour $h_i(\mathbf{x}) = 0$ must be zero.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If we have a point $\mathbf{x}$ that satisfies a single constraint $h(\mathbf{x}) = 0$, we are only allowed to improve f by traversing in a direction perpendicular to $\nabla h(\mathbf{x})$. For multiple constraints, we can only move a point $\mathbf{x}$ that satisfies all constraints in a direction perpendicular to all gradients $\nabla h_i(\mathbf{x})$. For $\mathbf{x}$ to be a critical point, f should not be locally improvable, which means that $\nabla f(\mathbf{x})$ must have zero component along any feasible direction. Because feasible directions are orthogonal to all constraint gradients, $\nabla f(\mathbf{x})$ must lie entirely within the span of the constraint gradients $\nabla h_i(\mathbf{x})$. In other words, $\nabla f(\mathbf{x})$ must be a linear combination:

$$\nabla f(\mathbf{x}) = \sum_i \lambda_i \nabla h_i(\mathbf{x}) \tag{10.17}$$

We can define a Lagrangian with multipliers associated with ℓ constraints:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i=1}^{\ell} \lambda_i h_i(\mathbf{x}) = f(\mathbf{x}) - \boldsymbol{\lambda}^\top \mathbf{h}(\mathbf{x}) \tag{10.18}$$

Points that satisfy $\nabla \mathcal{L} = \mathbf{0}$ will satisfy equation (10.17) and all the equality constraints.

10.7 *Inequality Constraints*

Consider a problem with a single inequality constraint:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g(\mathbf{x}) \leq 0 \end{aligned} \tag{10.19}$$

A solution may be either on the constraint boundary (figure 10.7) or in the interior of the feasible region (figure 10.8). We discuss these two cases separately.

If the solution lies at the constraint boundary where $g(\mathbf{x}) = 0$, then we know that the Lagrange condition holds

$$\nabla f + \mu \nabla g = \mathbf{0} \tag{10.20}$$

For inequality constraints, we use μ for the Lagrange multiplier. In contrast with equality constraints, we know that ∇g points in the opposite direction of ∇f; otherwise, we can move further into the feasible region and also decrease the objective. Hence, $\mu \geq 0$ if the solution lies on the constraint boundary. When this occurs, the constraint is considered *active*.

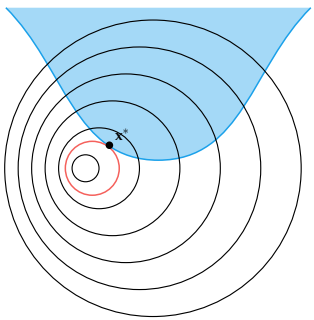


Figure 10.7. An active inequality constraint. The corresponding contour line is shown in red.

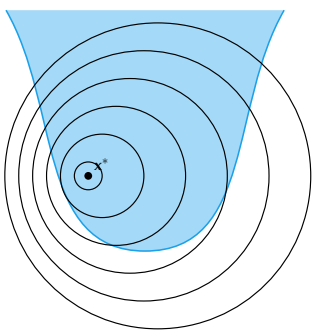


Figure 10.8. An inactive inequality constraint.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If the solution to the problem does not lie at the constraint boundary, then the constraint is *inactive*. As with unconstrained optimization, ∇f will be zero at a solution. In this case, equation (10.20) will hold by setting μ to zero.

We could optimize a problem with an inequality constraint by introducing an infinite step penalty for infeasible points:[8]

$$f_{\infty\text{-step}}(\mathbf{x}) = f(\mathbf{x}) + \infty(g(\mathbf{x}) > 0) \tag{10.21}$$

Unfortunately, $f_{\infty\text{-step}}$ is inconvenient to optimize. It is discontinuous and nondifferentiable. Search routines obtain no directional information to steer themselves toward feasibility. We can instead use a linear penalty $\mu g(\mathbf{x})$, which forms a lower bound on $\infty(g(\mathbf{x}) > 0)$ and penalizes the objective toward feasibility when $\mu > 0$. This linear penalty is shown in figure 10.9.

We can use this linear penalty to construct what is called the *generalized Lagrangian* for inequality constraints

$$\mathcal{L}(\mathbf{x}, \mu) = f(\mathbf{x}) + \mu g(\mathbf{x}) \tag{10.22}$$

We can recover $f_{\infty\text{-step}}$ by maximizing with respect to μ

$$f_{\infty\text{-step}}(\mathbf{x}) = \underset{\mu \geq 0}{\text{maximize}}\, \mathcal{L}(\mathbf{x}, \mu) \tag{10.23}$$

For any infeasible $\mathbf{x}$ we get infinity and for any feasible $\mathbf{x}$ we get $f(\mathbf{x})$.

The new optimization problem is thus

$$\underset{\mathbf{x}}{\text{minimize}}\, \underset{\mu \geq 0}{\text{maximize}}\, \mathcal{L}(\mathbf{x}, \mu) \tag{10.24}$$

This reformulation is known as the *primal* problem. Optimizing the primal problem requires finding critical points $\mathbf{x}^*$ such that:

1. $g(\mathbf{x}^*) \leq 0$
   The point is feasible.

2. $\mu \geq 0$
   The penalty must point in the right direction. This requirement is sometimes called *dual feasibility*. In this context, μ is referred to as a *dual variable* in the optimization. Further discussion of the notion of duality is contained in the next chapter.

[8] The function $\infty(\cdot)$ outputs ∞ if the input is true and zero otherwise.

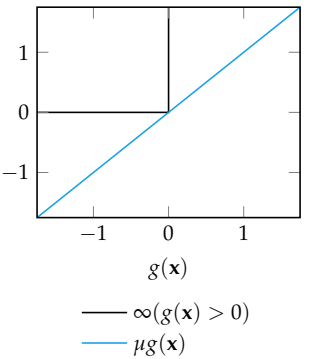


Figure 10.9. The linear function $\mu g(\mathbf{x})$ is a lower bound to the infinite step penalty when $\mu \geq 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3. $\mu g(\mathbf{x}^*) = 0$
   A feasible point on the boundary will have $g(\mathbf{x}) = 0$, whereas a feasible point with $g(\mathbf{x}) < 0$ will have $\mu = 0$ to recover $f(\mathbf{x}^*)$ from the Lagrangian.

4. $\nabla f(\mathbf{x}^*) + \mu \nabla g(\mathbf{x}^*) = \mathbf{0}$
   When the constraint is active, we require that the contour lines of f and g be aligned, which is equivalent to saying that their gradients be aligned. When the constraint is inactive, our optimum will have $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $\mu = 0$.

These four requirements can be generalized to optimization problems with any number of equality and inequality constraints:[9]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{10.25}$$

where each component of $\mathbf{g}$ is an inequality constraint and each component of $\mathbf{h}$ is an equality constraint. The four conditions are called the *KKT conditions*.[10]

1. **Feasibility:** The constraints are all satisfied:

$$\mathbf{g}(\mathbf{x}^*) \leq \mathbf{0} \tag{10.26}$$

$$\mathbf{h}(\mathbf{x}^*) = \mathbf{0} \tag{10.27}$$

2. **Dual feasibility:** Penalties are toward feasibility:

$$\boldsymbol{\mu} \geq \mathbf{0} \tag{10.28}$$

3. **Complementary slackness:** For each constraint, either $g_i(\mathbf{x}^*)$ is zero or the associated Lagrange multiplier μ_i is zero:[11]

$$\boldsymbol{\mu} \odot \mathbf{g} = \mathbf{0} \tag{10.29}$$

   A constraint or Lagrange multiplier is considered *tight* if it is equal to zero and *slack* if it is not.

4. **Stationarity:** The gradient of the objective lies in the span of the gradients of the active constraints:[12]

$$\nabla f(\mathbf{x}^*) + \sum_i \mu_i \nabla g_i(\mathbf{x}^*) + \sum_j \lambda_j \nabla h_j(\mathbf{x}^*) = \mathbf{0} \tag{10.30}$$

[9] If $\mathbf{u}$ and $\mathbf{v}$ are vectors of the same length, then we say $\mathbf{u} \leq \mathbf{v}$ when $u_i \leq v_i$ for all i. We define $\geq$, $<$, and $>$ similarly for vectors.

[10] Named after the American mathematician Harold W. Kuhn (1925–2014) and Canadian mathematician Albert W. Tucker (1905–1995) who published the conditions in 1951. It was later discovered that the American mathematician William Karush (1917–1997) studied these conditions in an unpublished master's thesis in 1939. A historical perspective is provided by T. H. Kjeldsen, "A Contextualized Historical Analysis of the Kuhn-Tucker Theorem in Nonlinear Programming: The Impact of World War II," *Historia Mathematica*, vol. 27, no. 4, pp. 331–361, 2000.

[11] The operation $\mathbf{a} \odot \mathbf{b}$ is the element-wise product between vectors $\mathbf{a}$ and $\mathbf{b}$.

[12] Since the sign of λ_j is not restricted, we can reverse the sign for the equality constraints from the method of Lagrange multipliers.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

These four conditions are first-order necessary conditions for optimality for problems with smooth constraints.[13] Just as with the conditions for unconstrained optimization, we must verify that critical points are actually minima.

[13] The second-order sufficient condition introduced in section 1.6.2 for local optimality in unconstrained problems can be generalized to constrained problems by analyzing the positive definiteness of the Lagrangian.

10.8 Slack Variables

A *slack variable* is a design variable introduced to an optimization problem that converts an inequality constraint into an equality constraint. We can make the following transformation by introducing a slack variable s:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && g(\mathbf{x}) \leq 0 \end{aligned} \quad \Rightarrow \quad \begin{aligned} &\underset{\mathbf{x},s}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && g(\mathbf{x}) + s = 0 \\ &&& s \geq 0 \end{aligned} \tag{10.31}$$

The slack variable is constrained to be nonnegative. When solved, the value of s^* is the amount by which the original inequality constraint function $g(\mathbf{x})$ can increase and remain feasible. The value of s^* thus represents the available slack in the constraint. If $s^* = 0$, then the original inequality constraint is active, and if $s^* > 0$, then the original inequality constraint is inactive. Introducing a slack variable increases the number of design variables, but the added problem complexity can sometimes lead to structure exploitable by optimization algorithms.[14]

[14] We will discuss such algorithms in chapters 12 and 14.

10.9 Penalty Methods

We can use *penalty methods* to convert constrained optimization problems into unconstrained optimization problems by adding penalty terms to the objective function, allowing us to use the methods developed in previous chapters.

Consider a general optimization problem:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ &&& \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{10.32}$$

A simple penalty method counts the violated constraints:

$$p_{\text{count}}(\mathbf{x}) = \sum_i (g_i(\mathbf{x}) > 0) + \sum_j (h_j(\mathbf{x}) \neq 0) \tag{10.33}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

which results in the unconstrained problem that penalizes infeasibility

$$\underset{\mathbf{x}}{\text{minimize}}\; f(\mathbf{x}) + \rho \cdot p_{\text{count}}(\mathbf{x}) \tag{10.34}$$

where $\rho > 0$ adjusts the penalty magnitude. Figure 10.10 shows an example.

Penalty methods start with an initial point $\mathbf{x}$ and a small value for ρ. The unconstrained optimization problem equation (10.34) is then solved. The resulting design point is then used as the starting point for another optimization with an increased penalty. We continue with this procedure until the resulting point is feasible, or a maximum number of iterations has been reached. Algorithm 10.2 provides an implementation.

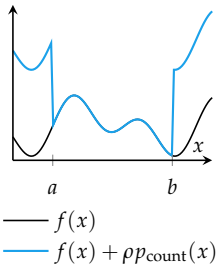


Figure 10.10. The original and count-penalized objective functions for minimizing f subject to $x \in [a, b]$.

```
function penalty_method(f, p, x, k_max; ρ=1.0, γ=2.0)
    for k in 1 : k_max
        x = minimize(x -> f(x) + ρ*p(x), x)
        ρ *= γ
        if p(x) == 0
            return x
        end
    end
    return x
end
```

Algorithm 10.2. The penalty method for objective function `f`, penalty function `p`, initial point `x`, number of iterations `k_max`, initial penalty `ρ > 0`, and penalty multiplier `γ > 1`. The function `minimize` can be any unconstrained method.

This penalty will preserve the problem solution for large values of ρ, but it introduces a sharp discontinuity. Points not inside the feasible set lack gradient information to guide the search toward feasibility.

We can use *quadratic penalties* to produce a smooth objective function (figure 10.11):

$$p_{\text{quadratic}}(\mathbf{x}) = \sum_i \max(g_i(\mathbf{x}), 0)^2 + \sum_j h_j(\mathbf{x})^2 \tag{10.35}$$

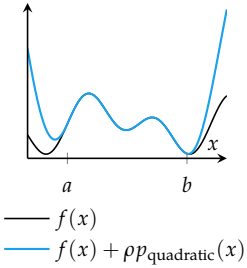


Figure 10.11. Using a quadratic penalty function for minimizing f subject to $x \in [a, b]$.

Quadratic penalties close to the constraint boundary are very small and may require ρ to approach infinity before the solution ceases to violate the constraints. It is common to add a small constant to $g_i(\mathbf{x})$ in the penalty function above to push the search toward the interior of the feasible region.

It is also possible to mix a count and a quadratic penalty function (figure 10.12):

$$p_{\text{mixed}}(\mathbf{x}) = \rho_1 p_{\text{count}}(\mathbf{x}) + \rho_2 p_{\text{quadratic}}(\mathbf{x}) \tag{10.36}$$

Such a penalty mixture both provides a clear boundary between the feasible region and the infeasible region and can provide gradient information to the solver.

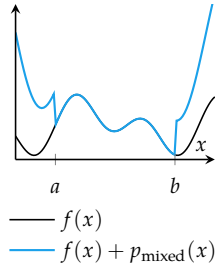


Figure 10.12. Using a mixed penalty function for minimizing f subject to $x \in [a, b]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Figure 10.13 shows the progress of the penalty function as ρ is increased. Quadratic penalty functions cannot ensure feasibility as discussed in example 10.5.

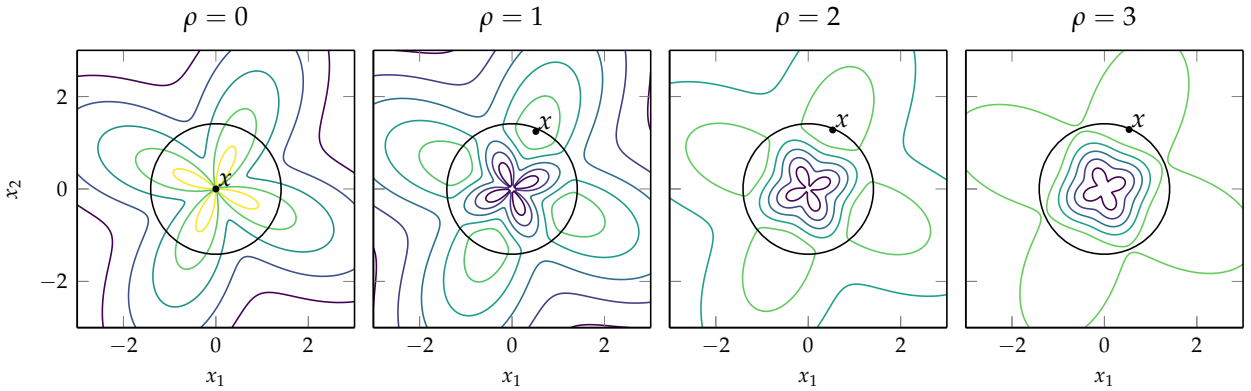


Figure 10.13. The penalty method applied to the flower function, appendix B.4, and the circular constraint $x_1^2 + x_2^2 \geq 2$.

10.10 Method of Multipliers

The *method of multipliers*, also known as the *augmented Lagrange method*, combines the quadratic penalty method with the linear penalties associated with Lagrange multipliers.[15] Unlike the penalty method, where ρ must sometimes approach infinity before a feasible solution is found, the method of multipliers will work with smaller values of ρ. It uses both a quadratic and a linear penalty for each constraint.

For an optimization problem with equality constraints $\mathbf{h}(\mathbf{x}) = \mathbf{0}$, the penalty function is[16]

$$p_{\text{Lagrange}}(\mathbf{x}) = \frac{1}{2}\rho \sum_i (h_i(\mathbf{x}))^2 + \sum_i \lambda_i h_i(\mathbf{x}) \tag{10.37}$$

In addition to increasing ρ with each iteration, the linear penalty vector is updated according to[17]

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \rho \mathbf{h}(\mathbf{x}^{(k+1)}) \tag{10.38}$$

Algorithm 10.3 provides an implementation.

[15] This method dates back to at least the 1960s. M. R. Hestenes, "Multiplier and Gradient Methods," *Journal of Optimization Theory and Applications*, vol. 4, no. 5, pp. 303–320, 1969.

[16] This method can be generalized to inequality constraints. D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996.

[17] Exercise 10.15 provides a justification for this update.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 10.5. An example showing how quadratic penalties cannot ensure feasibility.

Consider the problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x \\ \text{subject to} \quad & x \geq 5 \end{aligned}$$

using a quadratic penalty function.

The unconstrained objective function is

$$f(x) = x + \rho \max(5 - x, 0)^2$$

The minimum of the unconstrained objective function is

$$x^* = 5 - \frac{1}{2\rho}$$

While the minimum of the constrained optimization problem is clearly $x = 5$, the minimum of the penalized optimization problem merely approaches $x = 5$, requiring an infinite penalty to achieve feasibility.

Algorithm 10.3. The method of multipliers for objective function `f`, equality constraint function `h`, initial point `x`, number of iterations `k_max`, initial penalty scalar `ρ > 0`, and penalty multiplier `γ > 1`. The function `minimize` can be any minimization method.

```
function method_of_multipliers(f, h, x, k_max; ρ=1.0, γ=2.0)
    λ = zero(h(x))
    for k in 1 : k_max
        p(x) = ρ/2*sum(h(x).^2) + λ⋅h(x)
        x = minimize(x -> f(x) + p(x), x)
        λ += ρ*h(x)
        ρ *= γ
    end
    return x
end
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

10.11 Interior Point Methods

Interior point methods (algorithm 10.4), sometimes referred to as *barrier methods*, are optimization methods that ensure that the search points always remain feasible.[18] Interior point methods that are stopped early due to time or processing constraints can produce nearly optimal, though feasible, design points. They use a barrier function that approaches infinity as one approaches a constraint boundary. This barrier function $p_{\text{barrier}}(\mathbf{x})$ must satisfy several properties:

[18] A. S. Nemirovski and M. J. Todd, "Interior-Point Methods for Optimization," *Acta Numerica*, vol. 17, pp. 191–234, 2008.

1. $p_{\text{barrier}}(\mathbf{x})$ is continuous
2. $p_{\text{barrier}}(\mathbf{x})$ is nonnegative ($p_{\text{barrier}}(\mathbf{x}) \geq 0$) in the feasible region
3. $p_{\text{barrier}}(\mathbf{x})$ approaches infinity as $\mathbf{x}$ approaches any constraint boundary

Some examples of barrier functions include the *inverse barrier*:

$$p_{\text{barrier}}(\mathbf{x}) = -\sum_i \frac{1}{g_i(\mathbf{x})} \tag{10.39}$$

and the *log barrier*:[19]

$$p_{\text{barrier}}(\mathbf{x}) = -\sum_i \begin{cases} \log(-g_i(\mathbf{x})) & \text{if } g_i(\mathbf{x}) \geq -1 \\ 0 & \text{otherwise} \end{cases} \tag{10.40}$$

[19] The log barrier tends to be the barrier function of choice. It is affine-invariant, meaning that rescaling a g_i or applying an affine change of variables does not affect the direction obtained when solving the resulting problem using Newton's method.

A problem with inequality constraints can be transformed into an unconstrained optimization problem

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) + \frac{1}{\rho} p_{\text{barrier}}(\mathbf{x}) \tag{10.41}$$

When ρ is increased, the penalty for approaching the boundary decreases (figure 10.14).

Special care must be taken such that line searches do not leave the feasible region. Line searches $f(\mathbf{x} + \alpha\mathbf{d})$ are constrained to the interval $\alpha = [0, \alpha_u]$, where α_u is the step to the nearest boundary. In practice, α_u is chosen such that $\mathbf{x} + \alpha\mathbf{d}$ is just inside the boundary to avoid the boundary singularity.

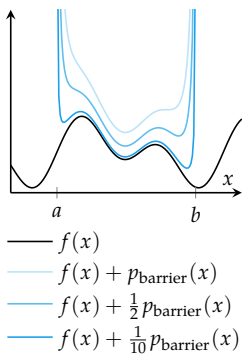


Figure 10.14. The interior point method with an inverse barrier for minimizing f subject to $x \in [a, b]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function interior_point_method(f, p, x; ρ=1.0, γ=2.0, ϵ=0.001)
    δ = Inf
    while δ > ϵ
        x′ = minimize(x -> f(x) + p(x)/ρ, x)
        δ = norm(x′ - x)
        x = x′
        ρ *= γ
    end
    return x
end
```

Algorithm 10.4. The interior point method for objective function `f`, barrier function `p`, initial point `x`, initial penalty `ρ > 0`, penalty multiplier `γ > 1`, and stopping tolerance `ϵ > 0`.

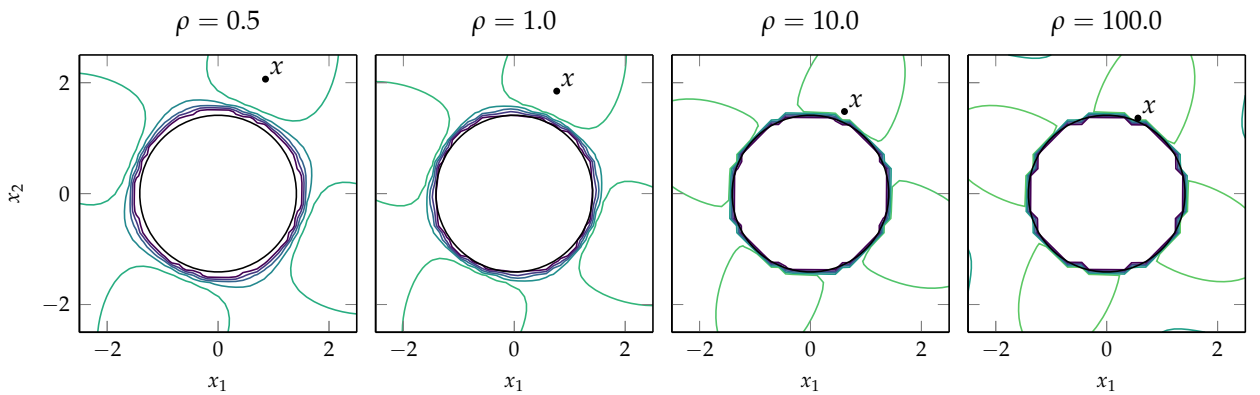


Figure 10.15. The interior point method with the inverse barrier applied to the flower function (appendix B.4) with the constraint $x_1^2 + x_2^2 \geq 2$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Like the penalty method, the interior point method begins with a low value for ρ and slowly increases it until convergence. The interior point method is typically terminated when the difference between subsequent points is less than a certain threshold. Figure 10.15 shows the effect of incrementally increasing ρ.

The interior point method requires a strictly feasible point from which to start the search. One method for finding such an interior point is to run the interior point method itself on a related problem with an additional variable s:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned} \quad \Rightarrow \quad \begin{aligned} &\underset{\mathbf{x},s}{\text{minimize}} && s \\ &\text{subject to} && \mathbf{g}(\mathbf{x}) \leq s\mathbf{1} \end{aligned} \tag{10.42}$$

This s serves as an upper bound on all inequality constraints. A feasible pair $(\mathbf{x}, s)$ with a negative value for s must have a design $\mathbf{x}$ that lies in the interior of the feasible set. The barrier method can be used to iterate to such a pair starting from any initial $\mathbf{x}^{(0)}$ and a sufficiently large value for $s^{(0)}$ such that $s^{(0)} > \max_i g_i(\mathbf{x}^{(0)})$.

10.12 Summary

- Constraints are requirements on the design points that a solution must satisfy.
- Some constraints can be transformed or substituted into the problem to result in an unconstrained optimization problem.
- The objective function can also be optimized without taking constraints into account, instead repeatedly projecting the iterate to the nearest feasible point.
- Incorporation of Lagrange multipliers leads to the generalized Lagrangian and the necessary conditions for optimality under constraints.
- Penalty methods penalize infeasible solutions and often provide gradient information to the optimizer to guide infeasible points toward feasibility.
- Interior point methods use barrier functions to avoid leaving the feasible set.

10.13 Exercises

Exercise 10.1. Solve

$$\begin{aligned} &\underset{x}{\text{minimize}} && x \\ &\text{subject to} && x \geq 0 \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

using the quadratic penalty method with $\rho > 0$. Solve the problem in closed form.

Solution: First reformulate the problem as $f(x) = x + \rho \max(-x, 0)^2$ for which the derivative is

$$f'(x) = \begin{cases} 1 + 2\rho x & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

This unconstrained objective function can be solved by setting $f'(x) = 0$, which yields the solution $x^* = -\frac{1}{2\rho}$. Thus, as $\rho \to \infty$ we have that $x^* \to 0$.

Exercise 10.2. Solve the problem above using the count penalty method with $\rho > 1$ and compare it to the quadratic penalty method.

Solution: The problem is reformulated to $f(x) = x + \rho(x < 0)$. The unconstrained objective function is unbounded from below when ρ is finite and x approaches negative infinity. The correct solution is not found, whereas the quadratic penalty method is able to approach the correct solution.

Exercise 10.3. Suppose that we are solving a constrained problem with the penalty method. We notice that the iterates remain infeasible, and we decide to stop the algorithm. What can we do to improve our search?

Solution: You might try to increase the penalty parameter ρ. It is possible that ρ is too small and the penalty term is ineffective. In such cases, the iterates may be reaching an infeasible region where the function decreases faster than the penalty terms, causing the method to converge on an infeasible solution.

Exercise 10.4. Consider a simple univariate minimization problem where you minimize a function $f(x)$ subject to $x \geq 0$. Assume that we know that the constraint is active; that is, $x^* = 0$ where x^* is the minimizer. Also, assume that we know that $f'(x^*) > 0$. Show that solving the same problem with the penalty method

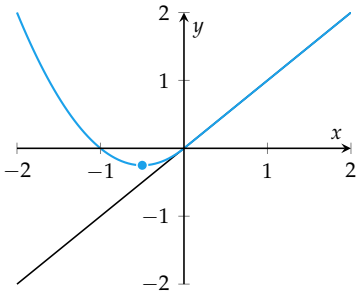
$$f(x) + (\min(x, 0))^2$$

yields an infeasible solution with respect to the original problem.

Solution: Let x_p^* be the solution to the unconstrained problem. Notice that x_p^* cannot be positive. Otherwise the penalty would be $\min(x_p^*, 0)^2 = 0$, which would imply that x_p^* is a solution to the original problem. Now, suppose $x_p^* = 0$. The first-order optimality conditions for unconstrained problems state that $f'(x_p^*) = 0$, again a contradiction. Thus, if a minimizer exists, it must be infeasible.

Below we show the infeasible minimizer when applying this penalty method for $f(x) = x$:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 10.5. What is the advantage of the method of multipliers compared to the quadratic penalty method?

Solution: It does not require a large penalty ρ to produce a feasible solution.

Exercise 10.6. When would you use the barrier method in place of the penalty method?

Solution: We would use the barrier method when iterates should remain feasible.

Exercise 10.7. Give an example of a smooth optimization problem, such that, for any quadratic penalty parameter $\rho > 0$, there exists a starting point $x^{(1)}$ for which the steepest descent method diverges in an infeasible region.

Solution: Consider the following:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x^3 \\ \text{subject to} \quad & x \geq 0 \end{aligned}$$

which is minimized for $x^* = 0$. Using the quadratic penalty method, we can recast it as

$$\underset{x}{\text{minimize}}\, x^3 + \rho(\min(x, 0))^2$$

For any finite ρ, the function remains unbounded from below as x becomes infinitely negative. Furthermore, as x becomes infinitely negative the function becomes infinitely steep. In other words, if we start the steepest descent method too far to the left, we have $x^3 + \rho x^2 \approx x^3$, and the penalty would be ineffective, and the steepest descent method will diverge.

Exercise 10.8. If we are using a quadratic penalty function with the steepest descent method, and it fails to find a feasible point even for large values of penalty parameter ρ, what can we do to improve the search?

Solution: We can set the objective to 0, and then optimize the resulting problem with a quadratic penalty function using steepest descent. Once we find a feasible point, we can then optimize the original problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 10.9. Implement projected descent for an affine inequality constraint:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \end{aligned}$$

and then use it to optimize Wheeler's Ridge (appendix B.7) for $a = 1.5$ with

$$\begin{bmatrix} 1.0 \\ 0.2 \end{bmatrix}^\top \mathbf{x} \leq 2.5 \qquad \begin{bmatrix} 0.5 \\ 0.7 \end{bmatrix}^\top \mathbf{x} \leq 2.5 \qquad \begin{bmatrix} 1.0 \\ -0.1 \end{bmatrix}^\top \mathbf{x} \geq 1.0 \qquad \begin{bmatrix} 0.3 \\ 0.8 \end{bmatrix}^\top \mathbf{x} \geq 1.5$$

and plot its progression from $x = [1.75, 1.75]$. You may call out to existing solvers in your implementation of `project_to_feasible_set`.

Solution: Affine constraints form convex sets, so the projection problem is convex, and can be solved by `Convex.jl`. This can be implemented as:

```
struct AffineConstraint
    A::Matrix{Float64}
    b::Vector{Float64}
end

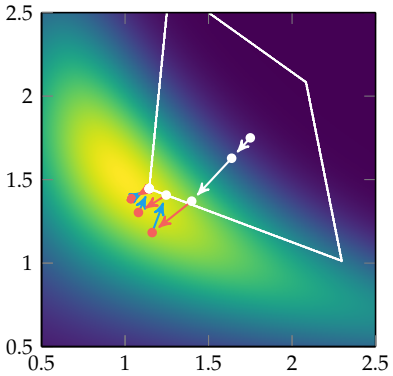
function project_to_feasible_set(X::AffineConstraint, x_in)
    x = Variable(length(x_in))
    problem = minimize(
        norm(x - x_in, 2),
        X.A*x ≤ X.b
    )
    solve!(problem, SCS.Optimizer, silent=true)
    return evaluate(x)
end
```

This method can then be used with projected descent (algorithm 10.1) and the gradient descent method (algorithm 5.1) to solve our problem. We can define the following:

```
a = 1.5
f = x -> -exp(-(x[1]*x[2] - a)^2 -(x[2]-a)^2)
∇f = x -> [2*exp(-(a-x[2])^2-(a-x[1]*x[2])^2) * x[2]*(-a+x[1]*x[2]),
             exp(-(a-x[2])^2-(a-x[1]*x[2])^2)*(-2*a*(1+x[1]) + 2*(1+x[1]^2)*x[2])]
x = [1.75,1.75]
M = GradientDescent(0.25)
X = AffineConstraint(
    [1.0  0.2
     0.5  0.7
    -1.0  0.1
    -0.3 -0.8], [2.5, 2.5, -1.0, -1.5])
```

Plotting this yields:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



We see the iterates (white) start in the interior of the feasible set, but eventually escape it as they head toward the unconstrained minimum. These iterates are then repeatedly projected back to the feasible set, and effectively walk along the constraint boundary until stopping at the feasible vertex closest to the unconstrained minimum.

Exercise 10.10. Solve the constrained optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sin\left(\frac{4}{x}\right) \\ \text{subject to} \quad & x \in [1, 10] \end{aligned}$$

using both the transform $x = t_{a,b}(\hat{x})$ introduced in equation (10.5) and a sigmoid transform for constraint bounds $x \in [a, b]$ defined by:

$$x = s_{a,b}(\hat{x}) = a + \frac{(b-a)}{1 + e^{-\hat{x}}}$$

Why is the $t_{a,b}$ transform better suited for optimizing our problem than the $s_{a,b}$ transform?

Solution: The problem is minimized at $x^* = 1$, which is at the constraint boundary. Solving with the $t_{a,b}$ transform yields the unconstrained objective function:

$$f_t(\hat{x}) = \sin\left(\frac{4}{5.5 + 4.5\frac{2\hat{x}}{1+\hat{x}^2}}\right)$$

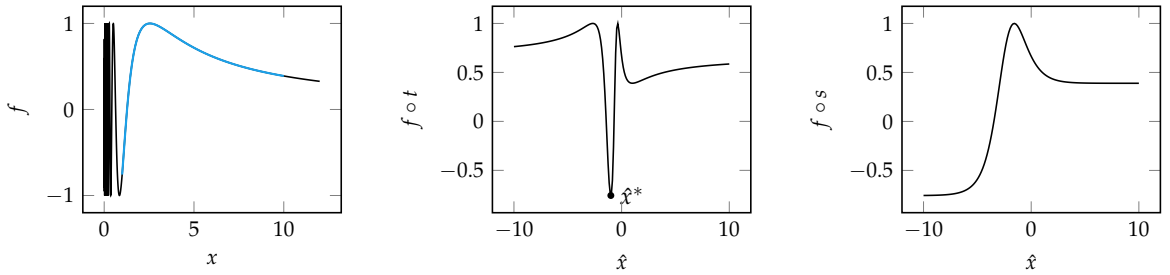
which has a single global minimum at $\hat{x} = -1$, correctly corresponding to x^*.

The $s_{a,b}$ transform has an unconstrained objective function:

$$f_s(\hat{x}) = \sin\left(\frac{4}{1 + \frac{9}{1+e^{-\hat{x}}}}\right)$$

Unfortunately, the lower bound on x is reached only as $\hat{x}$ approaches minus infinity. The unconstrained optimization problem obtained using the sigmoid transform does not have a solution, and the method fails to properly identify the solution of the original problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 10.11. Give an example of a quadratic objective function involving two design variables where the addition of a linear constraint results in a different optimum.

Solution: Minimize $x_1^2 + x_2^2$ subject to $x_1 \geq 1$.

Exercise 10.12. Suppose we want to minimize $x_1^3 + x_2^2 + x_3$ subject to the constraint that $x_1 + 2x_2 + 3x_3 = 6$. How might we transform this into an unconstrained problem with the same minimizer?

Solution: We can rearrange the constraint in terms of x_1:

$$x_1 = 6 - 2x_2 - 3x_3$$

and substitute the relation into the objective:

$$\underset{x_2, x_3}{\text{minimize}}\; x_2^2 + x_3 - (2x_2 + 3x_3 - 6)^3$$

Exercise 10.13. Suppose we want to minimize $-x_1 - 2x_2$ subject to the constraints $ax_1 + x_2 \leq 5$ and $x_1, x_2 \geq 0$. If a is a bounded constant, what range of values of a will result in an infinite number of optimal solutions?

Solution: To have infinitely many solutions, we need the boundary of the feasible region defined by $ax_1 + x_2 = 5$ to be aligned with the contour of the objective function $-x_1 - 2x_2$. Mathematically, we need $[a, 1] = \lambda[-1, -2]$. This can happen only when $a = 0.5$ and $\lambda = -0.5$.

Exercise 10.14. Consider using a penalty method to optimize

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 1 - x^2 \\ \text{subject to} \quad & |x| \leq 2 \end{aligned}$$

Optimization with the penalty method typically involves running several optimizations with increasing penalty weights. Impatient engineers may wish to optimize once using a very large penalty weight. Explain what issues are encountered for both the count penalty method and the quadratic penalty method.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The transformed objective function is $f(x) = 1 - x^2 + \rho p(x)$, where p is either a count penalty or a quadratic penalty:

$$p_{\text{count}}(x) = (|x| > 2) \qquad p_{\text{quadratic}}(x) = \max(|x| - 2, 0)^2$$

The count penalty method does not provide any gradient information to the optimization process. An optimization algorithm initialized outside of the feasible set will be drawn away from the feasible region because $1 - x^2$ is minimized by moving infinitely far to the left or right from the origin. The large magnitude of the count penalty is not the primary issue; small penalties can lead to similar problems.

The quadratic penalty method does provide gradient information to the optimization process, guiding searches toward the feasible region. For very large penalties, the quadratic penalty method will produce large gradient values in the infeasible region. In this problem, the partial derivative is:

$$\frac{\partial f}{\partial x} = -2x + \rho \begin{cases} 2(x-2) & \text{if } x > \quad 2 \\ 2(x+2) & \text{if } x < -2 \\ 0 & \text{otherwise} \end{cases}$$

For very large values of ρ, the partial derivative in the infeasible region is also large, which can cause problems for optimization methods. If ρ is not large, then infeasible points may not be sufficiently penalized, resulting in infeasible solutions.

Exercise 10.15. Write down the primal problem from equation (10.24) with a quadratic penalty function included. Derive an update equation for $\boldsymbol{\lambda}$ to solve the inner maximization problem for an updated iterate $\mathbf{x}^{(k+1)}$. Use gradient ascent for this update with a step factor of ρ.

Solution: The primal problem with a quadratic penalty function is

$$\underset{\mathbf{x}}{\text{minimize}}\, \underset{\boldsymbol{\lambda}}{\text{maximize}}\, f(\mathbf{x}) + \frac{1}{2}\rho \mathbf{h}(\mathbf{x})^\top \mathbf{h}(\mathbf{x}) + \boldsymbol{\lambda}^\top \mathbf{h}(\mathbf{x})$$

The update equation for $\boldsymbol{\lambda}$ with step factor ρ is

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \rho \mathbf{h}(\mathbf{x}^{(k+1)})$$

Exercise 10.16. Suppose we have an optimization problem with a single squared equality constraint:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & c^2(\mathbf{x}) = 0 \end{aligned}$$

where f and c have continuous partial derivatives. Why can we not determine the Lagrange multiplier λ using the method of Lagrange multipliers in this case?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The method of Lagrange multipliers enforces the condition:

$$\nabla f(\mathbf{x}) = \lambda \nabla h(\mathbf{x}) = 2\lambda c(\mathbf{x}) \nabla c(\mathbf{x})$$

We cannot determine λ from this equation because the gradient of the constraint is zero for all feasible points $\mathbf{x}$. However, we get necessary conditions for $\mathbf{x}^*$; namely, $\nabla f(\mathbf{x}^*) = 0$ and $c(\mathbf{x}^*) = 0$.

Exercise 10.17. How can the method of Lagrange multipliers be adapted to handle the optimization problem in the previous question?

Solution: Instead of using the constraint $c^2(\mathbf{x}) = 0$, we can use the equivalent constraint $c(\mathbf{x}) = 0$.

Exercise 10.18. Suppose we wish to solve an optimization problem using an interior point method. Could we find an initial feasible point by optimizing the quadratic penalty function?

$$\underset{\mathbf{x}}{\text{minimize}}\; p_{\text{quadratic}}(\mathbf{x})$$

Why might this be a good or a bad idea?

Solution: The quadratic penalty assigns zero penalty to the feasible set and a quadratic penalty outside of it. Hence, solutions to this quadratic penalty optimization problem are all feasible points in the original problem. In practice, optimization algorithms will tend to result in a design on or very close to the boundary of the feasible set when initialized with a point outside of the feasible set, but interior point methods require strictly feasible initial points.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

11 Duality

The previous chapter presented constrained minimization problems and derived the first-order necessary conditions for optimality, which included requirements on the dual variables that were introduced. This chapter elaborates on the concept of *duality*, which allows us to transform a constrained minimization problem into an analogous maximization problem.[1] The dual version of a problem provides a lower bound on the original problem. In some cases, duality provides a means of verification that a solution has been found. This chapter discusses a variety of algorithms that operate on the dual problem.

[1] Duality is covered in greater depth by S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

11.1 Dual Problem

In the previous chapter, we defined a general constrained optimization problem as an optimization problem with any number of equality and inequality constraints:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{11.1}$$

The *generalized Lagrangian* corresponding to this general constrained minimization problem is

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_i \mu_i g_i(\mathbf{x}) + \sum_j \lambda_j h_j(\mathbf{x}) \tag{11.2}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ are known as Lagrange multipliers or *dual variables*.

The *primal form* of the problem is the original optimization problem formulated using the generalized Lagrangian:

$$\underset{\mathbf{x}}{\text{minimize}} \underset{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}}{\text{maximize}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{11.3}$$

The interior maximization will drive the value to infinity if any constraints are violated, which ensures that the outer minimization will seek a feasible design. The primal problem is identical to the original problem and is just as difficult to optimize.

The *dual form* of the optimization problem reverses the order of the minimization and maximization in equation (11.3):

$$\underset{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}}{\text{maximize}} \underset{\mathbf{x}}{\text{minimize}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{11.4}$$

The *max-min inequality* states that for any function $f(\mathbf{a}, \mathbf{b})$:

$$\underset{\mathbf{a}}{\text{maximize}} \underset{\mathbf{b}}{\text{minimize}}\, f(\mathbf{a}, \mathbf{b}) \leq \underset{\mathbf{b}}{\text{minimize}} \underset{\mathbf{a}}{\text{maximize}}\, f(\mathbf{a}, \mathbf{b}) \tag{11.5}$$

The solution to the dual problem is thus a lower bound to the solution of the primal problem. That is, $d^* \leq p^*$, where d^* is the *dual value* and p^* is the *primal value*.

The inner minimization in the dual problem is often folded into a *dual function*,

$$\mathcal{D}(\boldsymbol{\mu}, \boldsymbol{\lambda}) = \underset{\mathbf{x}}{\text{minimize}}\, \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{11.6}$$

for notational convenience. The dual function is concave.[2] Gradient ascent can be used with this concave function to converge to the global maximum. Optimizing the dual problem is efficient whenever minimizing the Lagrangian with respect to $\mathbf{x}$ is efficient.

[2] For a fixed $\mathbf{x}$, $\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda})$ is affine in $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$. Because the minimum of a set of affine functions is concave, the dual function is concave. For a detailed overview, see S. Nash and A. Sofer, *Linear and Nonlinear Programming*. McGraw-Hill, 1996.

We know that $\text{maximize}_{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}}\, \mathcal{D}(\boldsymbol{\mu}, \boldsymbol{\lambda}) \leq p^*$. It follows that the dual function is always a lower bound on the primal problem (see example 11.1). For any $\boldsymbol{\mu} \geq \mathbf{0}$ and any $\boldsymbol{\lambda}$, we have $\mathcal{D}(\boldsymbol{\mu}, \boldsymbol{\lambda}) \leq p^*$. This property is known as *weak duality*.

The difference $p^* - d^*$ between the dual and primal values is called the *duality gap*. In some cases, the dual problem is guaranteed to have the same optimal value as the original problem, making the duality gap zero. This property is known as *strong duality*. Strong duality is guaranteed when the objective and constraints are convex and the constraints satisfy *Slater's condition*.[3] In such cases, duality can provide an alternative approach for optimizing our problem. Example 11.2 demonstrates this approach.

[3] Slater's condition and other conditions that guarantee zero duality gap are discussed by S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.1. The dual function is a lower bound of the primal problem.

Consider the optimization problem:

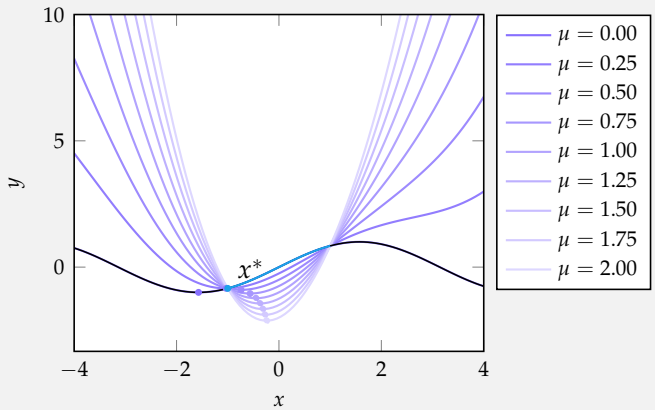
$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sin(x) \\ \text{subject to} \quad & x^2 \leq 1 \end{aligned}$$

The generalized Lagrangian is $\mathcal{L}(x, \mu) = \sin(x) + \mu(x^2 - 1)$, making the primal problem:

$$\underset{x}{\text{minimize}} \, \underset{\mu \geq 0}{\text{maximize}} \, \sin(x) + \mu(x^2 - 1)$$

and the dual problem:

$$\underset{\mu \geq 0}{\text{maximize}} \, \underset{x}{\text{minimize}} \, \sin(x) + \mu(x^2 - 1)$$



The objective function is plotted in black, with the feasible region indicated in blue. The minimum is at $x^* = -1$ with $p^* \approx -0.841$. The purple lines are the Lagrangian $\mathcal{L}(x, \mu)$ for different values of μ, each of which has a minimum lower than p^*. By inspection, we can see that strong duality also holds.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Consider the problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & x_1 + x_2 + x_1 x_2 \\ \text{subject to} \quad & x_1^2 + x_2^2 = 1 \end{aligned}$$

The Lagrangian is $\mathcal{L}(x_1, x_2, \lambda) = x_1 + x_2 + x_1 x_2 + \lambda(x_1^2 + x_2^2 - 1)$. We apply the method of Lagrange multipliers to obtain the necessary optimality conditions:

$$\frac{\partial \mathcal{L}}{\partial x_1} = 1 + x_2 + 2\lambda x_1 = 0 \quad \frac{\partial \mathcal{L}}{\partial x_2} = 1 + x_1 + 2\lambda x_2 = 0 \quad \frac{\partial \mathcal{L}}{\partial \lambda} = x_1^2 + x_2^2 - 1 = 0$$

Solving yields four potential solutions, and thus four critical points:

x_1	x_2	λ	$x_1 + x_2 + x_1 x_2$
-1	0	$1/2$	-1
0	-1	$1/2$	-1
$\frac{\sqrt{2}+1}{\sqrt{2}+2}$	$\frac{\sqrt{2}+1}{\sqrt{2}+2}$	$\frac{1}{2}\left(-1-\sqrt{2}\right)$	$\frac{1}{2} + \sqrt{2} \approx 1.914$
$\frac{\sqrt{2}-1}{\sqrt{2}-2}$	$\frac{\sqrt{2}-1}{\sqrt{2}-2}$	$\frac{1}{2}\left(-1+\sqrt{2}\right)$	$\frac{1}{2} - \sqrt{2} \approx -0.914$

We find that the two optimal solutions are $[-1, 0]$ and $[0, -1]$.

Substituting $u = x_1 + x_2$, $v = x_1 - x_2$ yields a dual function

$$\mathcal{D}(\lambda) = \underset{u,v}{\text{minimize}} \; \frac{1}{4}(2\lambda + 1)u^2 + u + \frac{1}{4}(2\lambda - 1)v^2 - \lambda$$

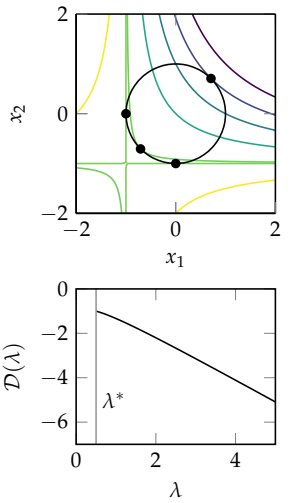
For $\lambda < 1/2$, the v^2 term is negative and the expression is unbounded below. For $\lambda = 1/2$, the v^2 term vanishes and the remaining expression $1/2u^2 + u - 1/2$ is minimized for $u = -1$. For $\lambda > 1/2$, both quadratic terms are positive, and the expression is minimized for

$$u = -\frac{2}{2\lambda + 1} \qquad v = 0 \quad \Rightarrow \quad x_1 = x_2 = -\frac{1}{2\lambda + 1}$$

The dual function is thus:

$$\mathcal{D}(\lambda) = \begin{cases} -\left(\frac{1}{2\lambda+1} + \lambda\right) & \lambda \geq \frac{1}{2} \\ -\infty & \text{otherwise} \end{cases}$$

Example 11.2. An example of Lagrangian duality applied to a problem with an equality constraint. The top figure shows the objective function contour and the constraint with the four critical points marked by scatter points. We have used the first-order necessary conditions, which while not generally sufficient for optimality, are necessary for optimality. The bottom figure shows the dual function.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

11.2 Primal-Dual Methods

Primal-dual methods build upon the interior point methods introduced in the previous chapter by simultaneously updating the values of the dual variables.[4] Such an approach can both speed convergence to a solution and also allow us to measure the duality gap. If the duality gap is ever sufficiently small, we know we can terminate.

[4] Primal-dual methods are covered in depth by S. J. Wright, *Primal-Dual Interior-Point Methods*. SIAM, 1997.

As discussed in section 10.11, interior point methods can be used to solve the inequality-constrained optimization problem

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned} \tag{11.7}$$

by approximating infinite discontinuities at the constraint boundaries with smooth barriers:

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) + \frac{1}{\rho} p_{\text{barrier}}(\mathbf{x}) \tag{11.8}$$

These problems are solved repeatedly with increasing values of ρ, yielding a sequence of designs that converge to a local minimum for the original problem. For the primal-dual method discussed here, we use a logarithmic barrier:

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) - \frac{1}{\rho} \sum_i \log(-g_i(\mathbf{x})) \tag{11.9}$$

Suppose we have a primal solution $\mathbf{x}^*$ to this barrier problem for a particular value of ρ. The primal solution will satisfy

$$\mathbf{0} = \nabla \left[f(\mathbf{x}^*) - \frac{1}{\rho} \sum_i \log(-g_i(\mathbf{x}^*)) \right] \tag{11.10}$$

$$= \nabla f(\mathbf{x}^*) + \frac{1}{\rho} \sum_i \frac{1}{-g_i(\mathbf{x}^*)} \nabla g_i(\mathbf{x}^*) \tag{11.11}$$

$$= \nabla f(\mathbf{x}^*) + \sum_i \frac{-1}{\rho g_i(\mathbf{x}^*)} \nabla g_i(\mathbf{x}^*) \tag{11.12}$$

If we set $\mu_i^* = -1/(\rho g_i(\mathbf{x}^*))$, then we produce a Lagrangian for the inequality-constrained optimization problem in equation (11.7):

$$\mathbf{0} = \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*) = \nabla f(\mathbf{x}^*) + \sum_i \mu_i^* \nabla g_i(\mathbf{x}^*) \tag{11.13}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The dual variable satisfies $\boldsymbol{\mu}^* \geq \mathbf{0}$ because $\mathbf{x}^*$ must be feasible. We are thus able to associate a dual value with every primal value obtained when iterating with the original interior point method.

Written another way, for a design $\mathbf{x}^*$ to be optimal in equation (11.9), there must exist a $\boldsymbol{\mu}^*$ such that

$$-\mu_i^* g_i(\mathbf{x}^*) = \frac{1}{\rho} \text{ for } i \text{ in } 1:m \tag{11.14}$$

This new equation is analogous to complementary slackness,[5] and approaches complementary slackness as ρ approaches infinity.

[5] Complementary slackness is derived in exercise 11.1. The derivation for this condition is similar.

The design $\mathbf{x}^*$ is not a solution to equation (11.7) in general. If there are m inequality constraints, then the duality gap for $\mathbf{x}^*$ and $\boldsymbol{\mu}^*$ is:

$$p^* - \mathcal{D}(\boldsymbol{\mu}^*) = f(\mathbf{x}^*) - \left(f(\mathbf{x}^*) + \sum_{i=1}^{m} \mu_i^* g_i(\mathbf{x}^*) \right) \tag{11.15}$$

$$= -\sum_{i=1}^{m} \frac{-1}{\rho g_i(\mathbf{x}^*)} g_i(\mathbf{x}^*) \tag{11.16}$$

$$= \frac{m}{\rho} \tag{11.17}$$

As $\rho \to \infty$, the duality gap approaches 0. As a result, $\mathbf{x}^*$ will approach the solution of our inequality-constrained optimization problem given in equation (11.7).

Our primal-dual method seeks to satisfy both stationarity and our alternative form of complementary slackness. We rearrange the equations such that they equal zero:

$$\begin{aligned} \nabla f(\mathbf{x}) + \sum_i \mu_i \nabla g_i(\mathbf{x}) &= \mathbf{0} \\ -\mu_i g_i(\mathbf{x}) - \frac{1}{\rho} &= 0 \quad \text{for } i \text{ in } 1:m \end{aligned} \tag{11.18}$$

The residual is as follows:

$$\mathbf{r}(\mathbf{x}, \boldsymbol{\mu}, \rho) = \begin{bmatrix} \nabla f(\mathbf{x}) + \sum_i \mu_i \nabla g_i(\mathbf{x}) \\ -\mu_1 g_1(\mathbf{x}) - \frac{1}{\rho} \\ \vdots \\ -\mu_m g_m(\mathbf{x}) - \frac{1}{\rho} \end{bmatrix} \tag{11.19}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The primal-dual method will seek $\mathbf{r} = \mathbf{0}$, varying both $\mathbf{x}$ and $\boldsymbol{\mu}$ over time, and increasing ρ as it proceeds. Its update is a line search in a primal-dual descent direction $\mathbf{d}_\mathbf{x}$ and $\mathbf{d}_\boldsymbol{\mu}$. We obtain a descent direction by applying Newton's method for root finding[6] to this objective. This calculation is implemented in algorithm 11.1 and is demonstrated in example 11.3.

[6] Newton's method for root finding solves $\mathbf{r}(\mathbf{x}, \boldsymbol{\mu}, \rho) = \mathbf{0}$ as opposed to Newton's method for minimization, covered in equation (6.6), which might be used to minimize $\|\mathbf{r}(\mathbf{x}, \boldsymbol{\mu}, \rho)\|$. The update for the root-finding method on $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ iterates using

$$\mathbf{x}' \leftarrow \mathbf{x} - \mathbf{J}(\mathbf{x})^{-1}\mathbf{f}(\mathbf{x})$$

where $\mathbf{J}$ is the Jacobian of $\mathbf{f}$.

```
function primal_dual_descent_direction(
    f, ∇f, Hf, gs, ∇gs, Hgs, ρ, x, μ)

    n, m = length(x), length(μ)
    r, J = zeros(n+m), zeros(n+m, n+m)
    r[1:n] = ∇f(x) + sum(μ[i]*∇gs[i](x) for i in 1:m)
    J[1:n, 1:n] = Hf(x) + sum(μ[i]*Hgs[i](x) for i in 1:m)
    for i in 1:m
        r[n+i] = -μ[i]*gs[i](x) - 1/ρ
        J[1:n,n+i] = ∇gs[i](x)
        J[n+i,1:n] = -μ[i]*∇gs[i](x)'
        J[n+i,n+i] = -gs[i](x)
    end
    d = pinv(J) * -r
    return (d_x=d[1:n], d_μ=d[n+1:end])
end
```

Algorithm 11.1. An algorithm for computing the descent direction for a primal-dual method. The algorithm takes an objective function `f`, its gradient `∇f`, its Hessian `Hf`; a vector of inequality constraint functions `gs`, their gradients `∇gs`, and their Hessians `Hgs`; an interior point scalar `ρ`, a design `x`, and dual variables `μ`.

As with previous interior point methods, this primal-dual method also requires that its iterates remain in the interior of the feasible set: $\mathbf{g}(\mathbf{x}) < \mathbf{0}$ and $\boldsymbol{\mu} > \mathbf{0}$. Assuming the initial values are in the feasible set, all we have to do to ensure feasibility is enforce feasibility during the line search along the descent direction.[7] This line search is a specialized form of backtracking line search (algorithm 4.3). If $\mathbf{d}$ is a valid descent direction, then there must exist a sufficiently small step size that satisfies both the sufficient decrease condition and keeps the iterates in the feasible set.

[7] The approach presented here is sufficient to ensure feasibility for convex constraints and objectives, but can still fail in the general case. This can happen, for example, if $\mathbf{g}$ oscillates across zero multiple times in the search direction.

Any negative components in $\mathbf{d}_\boldsymbol{\mu}$ could result in iterates violating $\boldsymbol{\mu} > \mathbf{0}$ if the steps are too large. Therefore, we bound an initial step size α

$$\alpha = \min\left(\alpha, -(1-\epsilon)\frac{\mu_i}{d_{\mu_i}}\right) \text{ for } i \text{ in } 1:m \text{ where } d_{\mu_i} < 0 \tag{11.20}$$

where ϵ is a small positive value that ensures $\mu_i > 0$ instead of $\mu_i \geq 0$. We then enforce $\mathbf{g}(\mathbf{x}) < \mathbf{0}$ by further reducing α by a reduction factor p until it is satisfied. Standard backtracking line search can be used to satisfy the sufficient decrease condition. Algorithm 11.2 provides an implementation, and figure 11.1 demonstrates how it works.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.3. Calculating the primal-dual descent direction for a simple optimization problem.

Suppose we wish to optimize

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & x^2 \\ \text{subject to} \quad & (x-3)^2 \leq 1 \end{aligned}$$

Iterations of our primal-dual method seek to satisfy:

$$\mathbf{r} = \begin{bmatrix} \nabla f(x) + \mu \nabla g(x) \\ -\mu g(\mathbf{x}) - \frac{1}{\rho} \end{bmatrix} = \begin{bmatrix} 2x + 2\mu(x-3) \\ -\mu(x-3)^2 + \mu - 1/\rho \end{bmatrix} = \mathbf{0}$$

The associated Jacobian is:

$$\mathbf{J} = \begin{bmatrix} \nabla^2 f(x) + \mu \nabla^2 g(x) & \nabla g(x) \\ -\mu \nabla g(x) & -g(x) \end{bmatrix} = \begin{bmatrix} 2 + 2\mu & 2(x-3) \\ -2\mu(x-3) & 1 - (x-3)^2 \end{bmatrix}$$

The descent direction is $\mathbf{d} = -\mathbf{J}^{-1}\mathbf{r}$ where d_1 is the descent direction for x and d_2 is the descent direction for μ.

11.3 Dual Ascent

Recall that the dual problem from equation (11.4) is always a concave optimization problem in $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$:

$$\underset{\boldsymbol{\mu} \geq \mathbf{0}, \boldsymbol{\lambda}}{\text{maximize}} \; \underset{\mathbf{x}}{\text{minimize}} \; \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{11.21}$$

When strong duality holds, an optimum $(\mathbf{x}^*, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*)$ will satisfy both:

$$\mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = \underset{\mathbf{x}}{\text{minimize}} \; \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) \tag{11.22}$$

and

$$\mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = \underset{\boldsymbol{\mu} \geq \mathbf{0}, \boldsymbol{\lambda}}{\text{maximize}} \; \mathcal{L}(\mathbf{x}^*, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{11.23}$$

This means the optimum can be approached in two ways: by minimizing the Lagrangian with respect to the design with appropriate fixed dual values or by maximizing the Lagrangian with respect to dual values with an appropriate design.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function primal_dual_interior_point(
    f, ∇f, Hf, gs, ∇gs, Hgs, ρ, x, μ;
    α_max=1.0, β=1e-4, γ=2, ϵ=0.01, p=0.9,
    res_min=1e-4, gap_min=1e-4)

    m = length(gs)
    r = (x, μ, ρ) -> [
        ∇f(x) + sum(μ[i]*∇gs[i](x) for i in 1:m)
        [-μ[i]*gs[i](x) - 1/ρ for i in 1:m]
    ]
    res = norm(r(x, μ, ρ))
    gap = sum(-μ[i]*gs[i](x) for i in 1:m)
    while res > res_min || gap > gap_min
        ρ *= γ
        d_x, d_μ = primal_dual_descent_direction(
                    f, ∇f, Hf, gs, ∇gs, Hgs, ρ, x, μ)
        # bound the step size to enforce μ > 0
        α = α_max
        for i in 1:m
            if d_μ[i] < 0
                α = min(α, -(1-ϵ)*μ[i] / d_μ[i])
            end
        end
        # multiply by a reduction factor until we have g(x) ≤ 0
        while any(g(x + α*d_x) > 0 for g in gs)
            α *= p
        end
        # continue to reduce until we have sufficient decrease
        while norm(r(x+α*d_x, μ+α*d_μ, ρ)) > (1-β*α)*res
            α *= p
        end
        x += d_x*α
        μ += d_μ*α
        res = norm(r(x, μ, ρ))
        gap = sum(-μ[i]*gs[i](x) for i in 1:m)
    end
    return (x, μ)
end
```

Algorithm 11.2. An implementation of a primal-dual method, which takes an objective function `f`, its gradient `∇f`, and its Hessian `Hf`; a vector of inequality constraint functions `gs`, their gradients `∇gs`, and their Hessians `Hgs`; an initial interior point scalar `ρ > 0`, a design `x`, and dual variables `μ`.

The algorithm proceeds until the norm of the residual is sufficiently decreased and the duality gap is sufficiently small. Not all problems have optimizers with zero duality gap, in which case a much larger duality gap threshold can be used.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

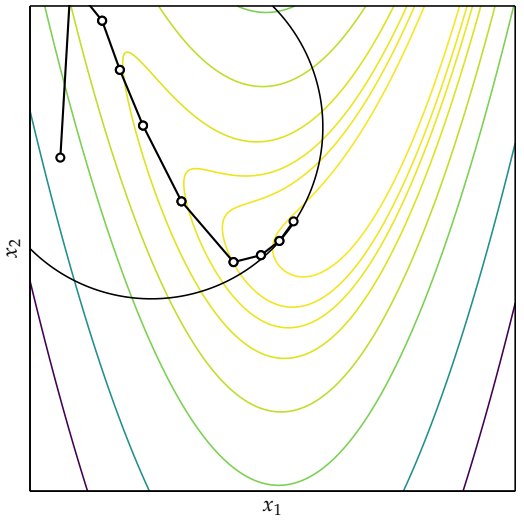


Figure 11.1. The primal dual interior point method applied to the Rosenbrock banana function with a circular constraint:

$$g(x) = (x_1 + 1)^2 + (x_2 - 1)^2 - 2$$

This run was initialized with $\mathbf{x} = [-1.75, 0.75]$ and $\boldsymbol{\mu} = [0.01]$ using $\rho = 1.0$ with growth factor 1.5.

In *dual ascent*, we alternate between solving for the primal values given the current values of the dual variables and applying an iteration of gradient ascent to the Lagrangian to update our dual variables. The aim is to move toward a $(\boldsymbol{\lambda}^*, \boldsymbol{\mu}^*)$ that maximizes the Lagrangian for $\mathbf{x}^*$. For an iterate $(\boldsymbol{\mu}^{(k)} \geq \mathbf{0}, \boldsymbol{\lambda}^{(k)})$, our update is:[8]

$$\mathbf{x}^{(k)} = \arg\min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}) \tag{11.24}$$

$$\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha^{(k)} \nabla_{\boldsymbol{\mu}} \mathcal{L}(\mathbf{x}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}) = \max\left(\boldsymbol{\mu}^{(k)} + \alpha^{(k)} \mathbf{g}(\mathbf{x}^{(k)}), \mathbf{0}\right) \tag{11.25}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \alpha^{(k)} \nabla_{\boldsymbol{\lambda}} \mathcal{L}(\mathbf{x}^{(k)}, \boldsymbol{\mu}^{(k)}, \boldsymbol{\lambda}^{(k)}) = \boldsymbol{\lambda}^{(k)} + \alpha^{(k)} \mathbf{h}(\mathbf{x}^{(k)}) \tag{11.26}$$

Dual ascent thus ascends on the dual problem. Algorithm 11.3 provides an implementation.

Dual ascent applies to problems with both equality and inequality constraints, does not require backtracking line search, and can include infeasible primal iterates. However, dual ascent may fail on problems even when they have zero duality gap. For example, problems with linear objectives can have unbounded $\mathbf{x}$ updates.

Dual ascent is related to the method of multipliers (section 10.10). Consider augmenting an equality constrained optimization problem with a quadratic penalty term:[9]

[8] The max here is elementwise to ensure $\boldsymbol{\mu} \geq \mathbf{0}$.

[9] This augmented problem has the same solution, since $\mathbf{h} = \mathbf{0}$ when solved.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function dual_ascent(f, g, h, x, μ, λ; α=1.0, γ=0.9, Δ_min=1e-4)
    L(x,μ,λ) = f(x) + μ⋅g(x) + λ⋅h(x)
    Δμ, Δλ = Inf, Inf
    while Δμ > Δ_min || Δλ > Δ_min
        x = minimize(x -> L(x,μ,λ), x)
        μ′ = max.(μ + α*g(x), 0)
        λ′ =      λ + α*h(x)
        Δμ, Δλ = norm(μ - μ′,2), norm(λ - λ′,2)
        μ, λ = μ′, λ′
        α *= γ
    end
    return (x, μ, λ)
end
```

Algorithm 11.3. The dual ascent method for solving a constrained optimization problem by ascending on the dual problem. The algorithm takes an objective function `f`, a vector-valued inequality constraint function `g`, a vector-valued equality constraint function `h`, and initial variables `x`, `μ`, and `λ`. The algorithm can additionally be parameterized by an initial step size `α`, a step size reduction factor `γ`, and a termination threshold `Δ_min` on the change in the dual variables.

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) + \frac{1}{2}\alpha\|\mathbf{h}(\mathbf{x})\|^2 \\ \text{subject to} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned} \tag{11.27}$$

Dual ascent on equation (11.27) reproduces the method of multipliers from section 10.10:

$$\mathbf{x}^{(k)} = \arg\min_{\mathbf{x}} f(\mathbf{x}) + \boldsymbol{\lambda}^\top \mathbf{h}(\mathbf{x}) + \frac{1}{2}\alpha\|\mathbf{h}(\mathbf{x})\|^2 \tag{11.28}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \alpha\mathbf{h}(\mathbf{x}^{(k)}) \tag{11.29}$$

11.4 Alternating Direction Method of Multipliers

The *alternating direction method of multipliers* (*ADMM*) is a method that can often scale to very large or distributed problems.[10] Like conjugate gradient descent, ADMM can exhibit fast convergence with modest accuracy and produce practical results in a few steps. As with other local descent methods, ADMM is not guaranteed to converge to a global optimum on nonconvex problems. It is particularly useful when the problem can be split across multiple processes or machines. ADMM operates on problems of the form:

$$\begin{aligned} \underset{\mathbf{x}_1,\, \mathbf{x}_2}{\text{minimize}} \quad & f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) \\ \text{subject to} \quad & \mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 = \mathbf{b} \end{aligned} \tag{11.30}$$

[10] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011. Z. Lin, H. Li, and C. Fang, *Alternating Direction Method of Multipliers for Machine Learning*. Springer, 2022. E. K. Ryu and W. Yin, *Large-Scale Convex Optimization: Algorithms and Analyses via Monotone Operators*. Cambridge University Press, 2023.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Notice that there are two design variables $\mathbf{x}_1$ and $\mathbf{x}_2$, which do not necessarily have the same dimension. The objective is separable in $\mathbf{x}_1$ and $\mathbf{x}_2$, and the constraint enforces an affine relationship between $\mathbf{x}_1$ and $\mathbf{x}_2$. Although this problem formulation may appear quite narrow with the additive decomposition and the linearity of the constraints, many problems—including general constrained optimization problems—can be reformulated in this form as discussed in the next section.

We apply the method of multipliers to equation (11.30), introducing a quadratic penalty term controlled by a scalar $\rho > 0$:

$$f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \frac{1}{2}\rho\|\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}\|_2^2 \tag{11.31}$$

This penalty term helps smooth the optimization problem with respect to the equality constraint, and it does not change the optimal value because all feasible points have 0 quadratic penalty.

We then construct the augmented Lagrangian:

$$\mathcal{L}_\rho(\mathbf{x}_1, \mathbf{x}_2, \boldsymbol{\lambda}) = f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \boldsymbol{\lambda}^\top(\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}) + \frac{1}{2}\rho\|\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}\|_2^2 \tag{11.32}$$

The method of multipliers would iterate according to:[11]

[11] Note the similarity to dual ascent.

$$(\mathbf{x}_1^{(k+1)}, \mathbf{x}_2^{(k+1)}) = \underset{\mathbf{x}_1, \mathbf{x}_2}{\arg\min}\, \mathcal{L}_\rho(\mathbf{x}_1, \mathbf{x}_2, \boldsymbol{\lambda}^{(k)}) \tag{11.33}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \rho(\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}) \tag{11.34}$$

The alternating direction method of multipliers instead alternates between minimizing $\mathbf{x}_1$ and $\mathbf{x}_2$ in every iteration:

$$\mathbf{x}_1^{(k+1)} = \underset{\mathbf{x}_1}{\arg\min}\, \mathcal{L}_\rho(\mathbf{x}_1, \mathbf{x}_2^{(k)}, \boldsymbol{\lambda}^{(k)}) \tag{11.35}$$

$$\mathbf{x}_2^{(k+1)} = \underset{\mathbf{x}_2}{\arg\min}\, \mathcal{L}_\rho(\mathbf{x}_1^{(k+1)}, \mathbf{x}_2, \boldsymbol{\lambda}^{(k)}) \tag{11.36}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \rho\left(\mathbf{A}_1\mathbf{x}_1^{(k+1)} + \mathbf{A}_2\mathbf{x}_2^{(k+1)} - \mathbf{b}\right) \tag{11.37}$$

This alternation of direction gives ADMM its name.

Algorithm 11.4 provides a basic implementation of ADMM. This implementation uses stopping criteria based on two residuals, the *primal residual* $\mathbf{r}$ that measures how much the equality constraint is violated:

$$\mathbf{r}^{(k+1)} = \mathbf{A}_1\mathbf{x}_1^{(k+1)} + \mathbf{A}_2\mathbf{x}_2^{(k+1)} - \mathbf{b} \tag{11.38}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and the *dual residual* $\mathbf{s}$:

$$\mathbf{s}^{(k+1)} = \rho \mathbf{A}_1^\top \mathbf{A}_2 \left(\mathbf{x}_2^{(k+1)} - \mathbf{x}_2^{(k)} \right) \tag{11.39}$$

The dual residual measures how close the design is to satisfying stationarity[12], in particular that

$$\nabla_{\mathbf{x}_1} \mathcal{L} = \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1) + \mathbf{A}_1^\top \boldsymbol{\lambda} = \mathbf{0} \tag{11.40}$$

The other stationarity requirement,

$$\nabla_{\mathbf{x}_2} \mathcal{L} = \nabla_{\mathbf{x}_2} f_2(\mathbf{x}_2) + \mathbf{A}_2^\top \boldsymbol{\lambda} = \mathbf{0} \tag{11.41}$$

is automatically enforced with every iteration.

[12] Exercise 11.4 derives this relation.

```
function admm(
    f1, f2, A1, A2, b, x1, x2;
    ρ=1, γ=2, ϵr=1e-3, ϵs=1e-3)

    λ = zeros(size(A2, 1))
    r = fill(Inf, size(A2, 1))
    s = fill(Inf, size(A1, 2))
    h(x1,x2) = A1*x1 + A2*x2 - b
    L(x1,x2,λ,ρ) = f1(x1)+f2(x2)+λ⋅h(x1,x2)+ρ/2*h(x1,x2)⋅h(x1,x2)
    while norm(r) > ϵr || norm(s) > ϵs
        x1′ = minimize(x1 -> L(x1, x2,λ,ρ), x1)
        x2′ = minimize(x2 -> L(x1′,x2,λ,ρ), x2)
        r  = h(x1′,x2′)
        λ += ρ*r
        s  = ρ*A1'*A2*(x2′ - x2)
        ρ *= γ
        x1, x2  = x1′, x2′
    end
    return x1
end
```

Algorithm 11.4. The alternating direction method of multipliers (ADMM) for the objective functions `f1` and `f2`; the equality constraint given by `A1`, `A2`, and `b`; initial design `x1` and `x2`; initial penalty scalar `ρ > 0` and penalty multiplier `γ > 1`; and convergence thresholds `ϵr` and `ϵs > 0`. This implementation uses warm starting, where the current values for `x1` and `x2` are used to initialize the minimization procedures. The current value is typically close to the minimizer, especially in later iterations, and using it tends to lead to faster convergence than using default initial values.

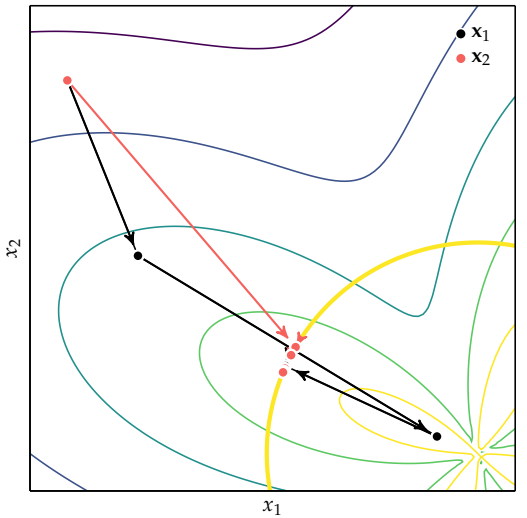
It can be beneficial to keep the primal and dual residuals close to one another as ADMM progresses.[13] Increasing the penalty parameter will tend to encourage reduction of the primal residual. A simple scheme can be used to automatically regulate the penalty parameter:

$$\rho^{(k+1)} = \begin{cases} \gamma^+ \rho^{(k)} & \text{if } \|\mathbf{r}^{(k)}\|_2 > \mu \|\mathbf{s}^{(k)}\|_2 \\ \gamma^- \rho^{(k)} & \text{if } \|\mathbf{s}^{(k)}\|_2 > \mu \|\mathbf{r}^{(k)}\|_2 \\ \rho^{(k)} & \text{otherwise} \end{cases} \tag{11.42}$$

[13] The suboptimality of the current point, $f_1(\mathbf{x}_1^{(k)}) + f_2(\mathbf{x}_2^{(k)}) - p^*$, can be shown to be bounded by $\|\boldsymbol{\lambda}^{(k)}\|_2 \|\mathbf{r}^{(k)}\|_2 + d \, \|\mathbf{s}^{(k)}\|_2$ for a $d \geq \|\mathbf{x}^{(k)} - \mathbf{x}^*\|_2$. Both residuals must approach zero for convergence to occur. If one residual reduces too quickly, it can slow down the rate at which the other can catch up.

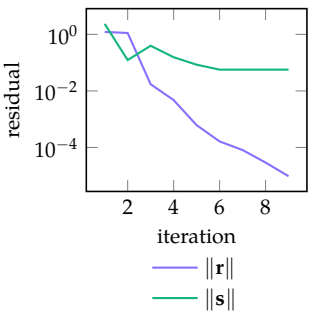
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where $\gamma^+ > 1$, $1 > \gamma^- > 0$, and $\mu > 1$.[14] This approach scales ρ when the primal residual is significantly larger than the dual residual, and shrinks it when the dual residual is significantly larger than the primal residual. These residuals are shown in figure 11.2, where ADMM is optimizing the flower function with a circle constraint.



[14] Suggested values include $\gamma_+ = 2$, $\gamma_- = 0.5$, and $\mu = 10$. B. S. He, H. Yang, and S. L. Wang, "Alternating Direction Method with Self-Adaptive Penalty Parameters for Monotone Variational Inequalities," *Journal of Optimization Theory and Applications*, vol. 106, no. 2, pp. 337–156, 2000.

Figure 11.2. ADMM applied to the flower function (appendix B.4) with the constraint $\|\mathbf{x}\|^2 = 2$. The transformation used to rewrite the problem into ADMM form will be covered in the next section. The designs $\mathbf{x}_1$ and $\mathbf{x}_2$ are initialized to the same value, but diverge because each optimizes different objectives. These quickly reconverge as the method progresses and ρ increases. The norms of the primal and dual residuals approach zero as the algorithm progresses.



The ADMM updates can often be simplified by converting the Lagrangian to *scaled form*. We replace $\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}$ with the residual $\mathbf{r}$ and write:

$$\mathcal{L}_\rho(\mathbf{x}_1, \mathbf{x}_2, \boldsymbol{\lambda}) = f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \boldsymbol{\lambda}^\top \mathbf{r} + \frac{1}{2}\rho\|\mathbf{r}\|_2^2 \tag{11.43}$$

$$= f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \boldsymbol{\lambda}^\top \mathbf{r} + \frac{1}{2}\rho\|\mathbf{r}\|_2^2 + \frac{1}{2\rho}\|\boldsymbol{\lambda}\|_2^2 - \frac{1}{2\rho}\|\boldsymbol{\lambda}\|_2^2 \tag{11.44}$$

$$= f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \boldsymbol{\lambda}^\top \mathbf{r} + \frac{1}{2}\rho\mathbf{r}^\top \mathbf{r} + \frac{1}{2\rho}\boldsymbol{\lambda}^\top \boldsymbol{\lambda} - \frac{1}{2\rho}\boldsymbol{\lambda}^\top \boldsymbol{\lambda} \tag{11.45}$$

$$= f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \frac{1}{2}\rho\left(\mathbf{r} + \frac{1}{\rho}\boldsymbol{\lambda}\right)^\top \left(\mathbf{r} + \frac{1}{\rho}\boldsymbol{\lambda}\right) - \frac{1}{2\rho}\boldsymbol{\lambda}^\top \boldsymbol{\lambda} \tag{11.46}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can introduce a norm and the *scaled dual variable* defined to be $\mathbf{u} = (1/\rho)\boldsymbol{\lambda}$:

$$\mathcal{L}_\rho(\mathbf{x}_1, \mathbf{x}_2, \boldsymbol{\lambda}) = f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \frac{1}{2}\rho\left\|\mathbf{r} + \frac{1}{\rho}\boldsymbol{\lambda}\right\|_2^2 - \frac{1}{2\rho}\|\boldsymbol{\lambda}\|_2^2 \tag{11.47}$$

$$= f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \frac{1}{2}\rho\|\mathbf{r} + \mathbf{u}\|_2^2 - \frac{1}{2}\rho\|\mathbf{u}\|_2^2 \tag{11.48}$$

$$= f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) + \frac{1}{2}\rho\|\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b} + \mathbf{u}\|_2^2 - \frac{1}{2}\rho\|\mathbf{u}\|_2^2 \tag{11.49}$$

Reformulating the ADMM updates with this scaled form yields:

$$\mathbf{x}_1^{(k+1)} = \arg\min_{\mathbf{x}_1} f_1(\mathbf{x}_1) + \frac{1}{2}\rho\left\|\mathbf{A}_1\mathbf{x}_1 - \mathbf{v}^{(k)}\right\|_2^2 \tag{11.50}$$

$$\mathbf{x}_2^{(k+1)} = \arg\min_{\mathbf{x}_2} f_2(\mathbf{x}_2) + \frac{1}{2}\rho\left\|\mathbf{A}_2\mathbf{x}_2 - \mathbf{w}^{(k)}\right\|_2^2 \tag{11.51}$$

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \mathbf{A}_1\mathbf{x}_1^{(k+1)} + \mathbf{A}_2\mathbf{x}_2^{(k+1)} - \mathbf{b} = \mathbf{u}^{(0)} + \sum_{j=1}^{k+1}\mathbf{r}^{(j)} \tag{11.52}$$

where $\mathbf{v}^{(k)} = -\mathbf{A}_2\mathbf{x}_2^{(k)} + \mathbf{b} - \mathbf{u}^{(k)}$ and $\mathbf{w}^{(k)} = -\mathbf{A}_1\mathbf{x}_1^{(k+1)} + \mathbf{b} - \mathbf{u}^{(k)}$. The scaled dual variable is identical to the running sum of the residuals.

11.5 ADMM Applications

Many problems that have been studied over the years can be expressed in ADMM form. This section presents a few examples.

11.5.1 General Constrained Optimization

Any constrained optimization problem can be reformulated in ADMM form. An optimization problem expressed as:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x} \in \mathcal{X} \end{aligned} \tag{11.53}$$

has the ADMM form:

$$\begin{aligned} &\underset{\mathbf{x}_1,\, \mathbf{x}_2}{\text{minimize}} && f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) \\ &\text{subject to} && \mathbf{x}_1 - \mathbf{x}_2 = \mathbf{0} \end{aligned} \tag{11.54}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

with $f_1 = f$ and

$$f_2(\mathbf{x}_2) = \begin{cases} 0 & \text{if } \mathbf{x}_2 \in \mathcal{X} \\ \infty & \text{otherwise} \end{cases} \tag{11.55}$$

We introduced $\mathbf{x}_2$ as a duplicate of the design vector $\mathbf{x}_1$. We continue to minimize the original objective function, but we also introduce a secondary objective $f_2(\mathbf{x}_2)$ to reflect whether or not our constraint is satisfied. The equality constraint uses $\mathbf{A}_1 = \mathbf{I}$, $\mathbf{A}_2 = -\mathbf{I}$, and $\mathbf{b} = \mathbf{0}$, resulting in a constraint that enforces equality between $\mathbf{x}_1$ and $\mathbf{x}_2$ when the problem is solved. This approach allows $\mathbf{x}_1$ to balance minimizing f and reaching consensus with $\mathbf{x}_2$, and $\mathbf{x}_2$ to balance feasibility and reaching consensus with $\mathbf{x}_1$. The penalty scalar ρ increases over time to draw the two designs together for eventual convergence.

Example 11.4 rewrites a constrained optimization problem in ADMM form and shows its $\mathbf{x}_1$- and $\mathbf{x}_2$-updates. The progression of ADMM on this problem is shown in figure 11.3.

The ADMM updates for problems in the form of equation (11.54) are:

$$\mathbf{x}_1^{(k+1)} = \arg\min_{\mathbf{x}_1} f_1(\mathbf{x}_1) + \boldsymbol{\lambda}^\top \mathbf{x}_1 + \frac{1}{2}\rho \left\| \mathbf{x}_1 - \mathbf{x}_2^{(k)} \right\|_2^2 \tag{11.56}$$

$$\mathbf{x}_2^{(k+1)} = \arg\min_{\mathbf{x}_2} f_2(\mathbf{x}_2) - \boldsymbol{\lambda}^\top \mathbf{x}_2 + \frac{1}{2}\rho \left\| \mathbf{x}_1^{(k+1)} - \mathbf{x}_2 \right\|_2^2 \tag{11.57}$$

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + \rho\left(\mathbf{x}_1^{(k+1)} - \mathbf{x}_2^{(k+1)}\right) \tag{11.58}$$

Both the $\mathbf{x}_1$- and $\mathbf{x}_2$-updates have an additional quadratic term that biases the solution toward the other design. This form of quadratic minimization is called *proximal minimization*,[15] with its name reflecting how the minimizer lies in the proximity of the other design. The larger that ρ is, the larger the bias. Proximal minimization is similar in principle to projected iterative methods (section 10.5), and has close ties to gradient descent as shown in example 11.5.

[15] N. Parikh and S. Boyd, "Proximal Algorithms," *Foundations and Trends in Optimization*, vol. 1, no. 3, pp. 127–239, 2014.

Proximal minimization is also particularly useful for the $\mathbf{x}_2$-update. As we recall, in equation (11.54), $f_2(\mathbf{x}_2)$ is an indicator function for a set membership constraint:

$$f_2(\mathbf{x}_2) = \begin{cases} 0 & \text{if } \mathbf{x}_2 \in \mathcal{X} \\ \infty & \text{otherwise} \end{cases} \tag{11.59}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.4. A simple constrained convex optimization problem rewritten in ADMM form.

Consider the optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \exp(x) \\ \text{subject to} \quad & (x-2)^2 \leq 1 \end{aligned}$$

This problem can be written in ADMM form as:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & \exp(x_1) + f_2(x_2) \\ \text{subject to} \quad & x_1 - x_2 = 0 \end{aligned}$$

where

$$f_2(x_2) = \begin{cases} 0 & \text{if } (x_2-2)^2 \leq 1 \\ \infty & \text{otherwise} \end{cases}$$

Its x_1-update is:

$$x_1^{(k+1)} = \arg\min_{x_1} \exp(x_1) + \lambda \cdot \left(x_1 - x_2^{(k)}\right) + \frac{1}{2}\rho\left(x_1 - x_2^{(k)}\right)^2$$

and its x_2-update is:

$$x_2^{(k+1)} = \arg\min_{x_2} f_2(x_2) + \lambda \cdot \left(x_1^{(k+1)} - x_2\right) + \frac{1}{2}\rho\left(x_1^{(k+1)} - x_2\right)^2$$

which is equivalent to:

$$\begin{aligned} x_2^{(k+1)} = \arg\min_{x_2} \quad & \lambda \cdot \left(x_1^{(k+1)} - x_2\right) + \frac{1}{2}\rho\left(x_1^{(k+1)} - x_2\right)^2 \\ \text{subject to} \quad & (x_2-2)^2 \leq 1 \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

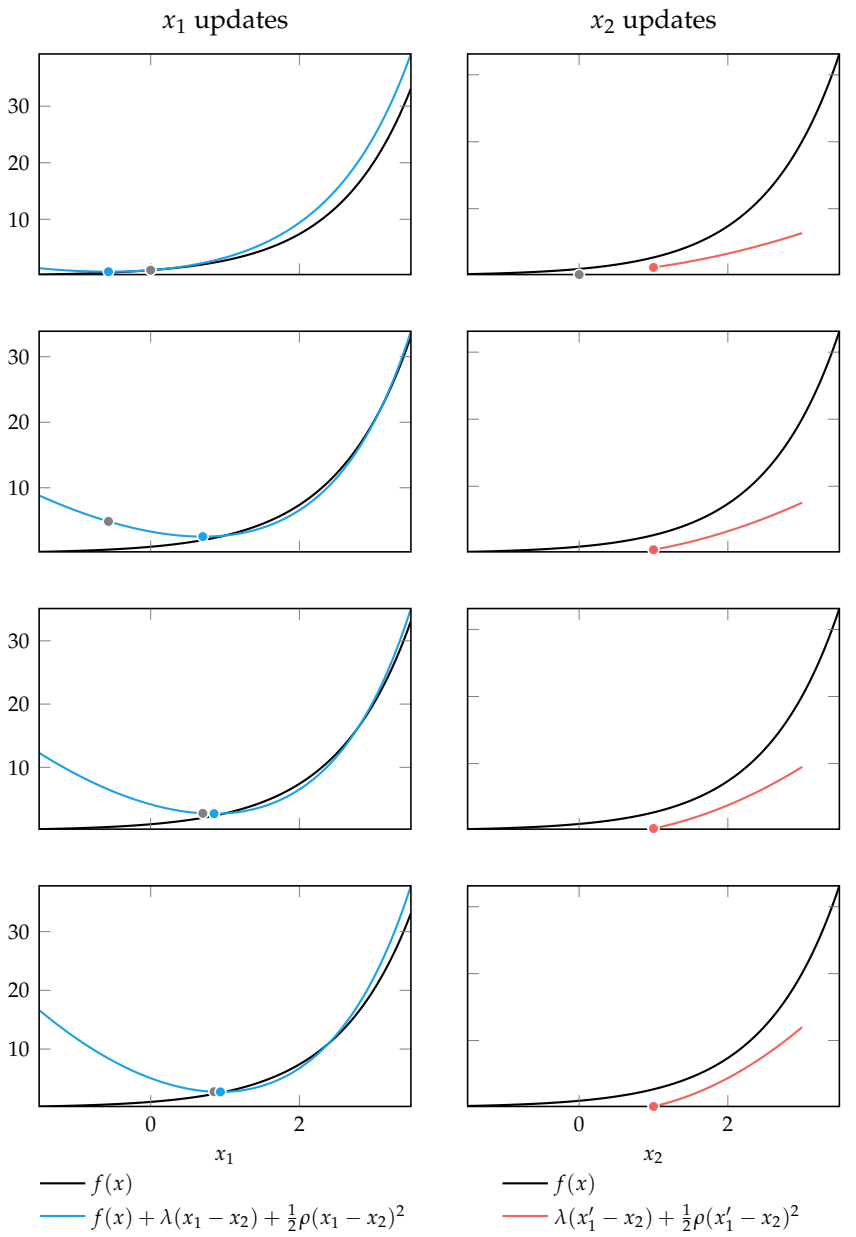


Figure 11.3. ADMM applied to the optimization problem in example 11.4. Iterations proceed top to bottom, with $\mathbf{x}_1$-updates depicted on the left and $\mathbf{x}_2$-updates depicted on the right. Each plot shows $\exp(x)$ in black along with the optimization function being optimized by the respective update. The gray dot shows the value at the start of the iteration, and the colored dot shows the updated value.

This progression started with $\rho = 1$ and used $\gamma = 1.5$. In this simple problem, x_2 snaps to 1, the closest feasible point, and then remains there throughout subsequent iterations. Notice how the $\mathbf{x}_1$-update is slowly contorted toward enforcing $x = 1$ as ρ increases, thereby converging to the optimal solution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.5. Proximal minimization applied to a first-order Taylor approximation yields a gradient step update.

Proximal minimization of a function f with respect to a point $\mathbf{p}$ produces a minimizer $\mathbf{x}$ in the proximity of $\mathbf{p}$:

$$\arg\min_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{2}\rho\|\mathbf{x} - \mathbf{p}\|_2^2$$

If f is differentiable, then we can approximate f with a first-order Taylor expansion about $\mathbf{p}$, where $f(\mathbf{x}) \approx f(\mathbf{p}) + \nabla f(\mathbf{p})^\top(\mathbf{x} - \mathbf{p})$. Proximal minimization applied to this approximation yields

$$\arg\min_{\mathbf{x}} \left(f(\mathbf{p}) + \nabla f(\mathbf{p})^\top(\mathbf{x} - \mathbf{p}) + \frac{1}{2}\rho\|\mathbf{x} - \mathbf{p}\|_2^2 \right) = \mathbf{p} - \frac{1}{\rho}\nabla f(\mathbf{p})$$

The update is a standard gradient step from $\mathbf{p}$ with step factor $1/\rho$.

If we ignore the $\boldsymbol{\lambda}^\top \mathbf{x}_2$ term, proximal minimization will find the feasible point closest to $\mathbf{x}_1^{(k+1)}$ as shown in example 11.6. In other words, $\mathbf{x}_2^{(k+1)}$ is the projection of $\mathbf{x}_1^{(k+1)}$ onto $\mathcal{X}$.

11.5.2 *Alternating Projections*

ADMM can be used to re-derive the *alternating projections* method.[16] We have two convex sets $\mathcal{X}_1$ and $\mathcal{X}_2$, and we want to find a design $\mathbf{x}$ that lies in both of them.

This problem can be written in ADMM form as:

$$\begin{aligned} &\underset{\mathbf{x}_1, \mathbf{x}_2}{\text{minimize}} && f_1(\mathbf{x}_1) + f_2(\mathbf{x}_2) \\ &\text{subject to} && \mathbf{x}_1 - \mathbf{x}_2 = \mathbf{0} \end{aligned} \tag{11.60}$$

where

$$f_i(\mathbf{x}_i) = \begin{cases} 0 & \text{if } \mathbf{x}_i \in \mathcal{X}_i \\ \infty & \text{otherwise} \end{cases} \tag{11.61}$$

[16] R. L. Dykstra, "An Algorithm for Restricted Least Squares Regression," *Journal of the American Statistical Association*, vol. 78, no. 384, pp. 837–842, 1983.

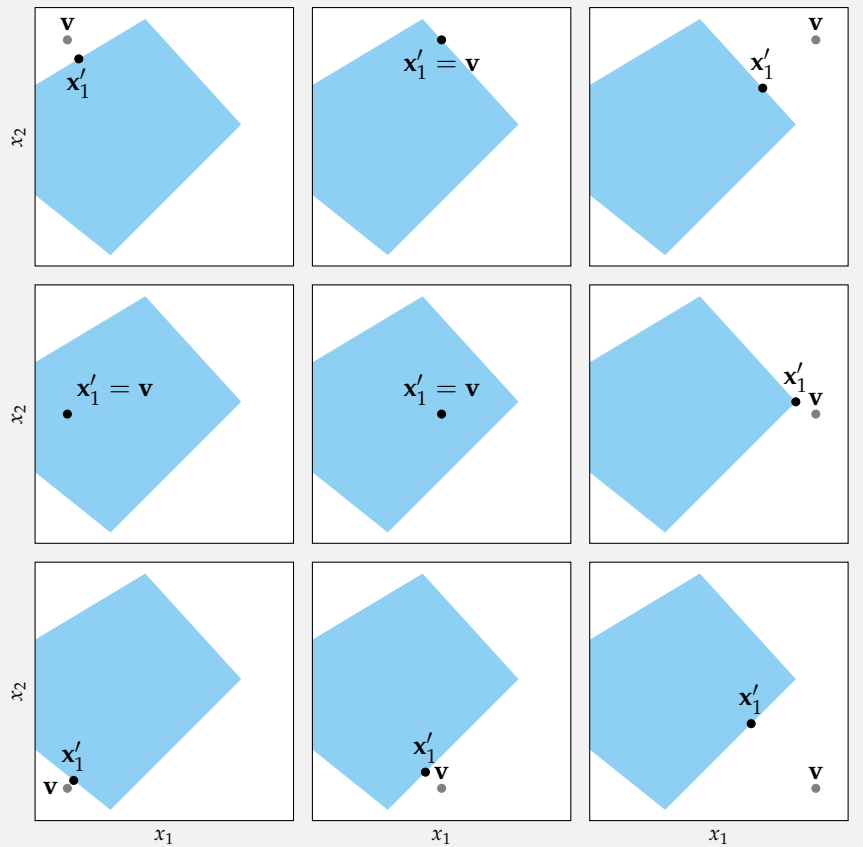
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.6. Proximal minimization can be used to enforce constraints.

Consider an objective function that is zero when $\mathbf{x}_1$ satisfies a convex inequality constraint and positive infinity otherwise. The $\mathbf{x}_1$-update is

$$\begin{aligned}\mathbf{x}_1' &= \arg\min_{\mathbf{x}_1} \infty(\mathbf{C}\mathbf{x}_1 > \mathbf{d}) + \frac{1}{2}\rho\|\mathbf{x}_1 - \mathbf{v}\|_2^2 \\ &= \arg\min_{\mathbf{x}_1 | \mathbf{C}\mathbf{x}_1 \leq \mathbf{d}} \frac{1}{2}\rho\|\mathbf{x}_1 - \mathbf{v}\|_2^2\end{aligned}$$

The output of proximal minimization for such a set is shown below for different values of $\mathbf{v}$. The feasible set $\mathcal{X} = \{\mathbf{x}_1 \mid \mathbf{C}\mathbf{x}_1 \leq \mathbf{d}\}$ is drawn in blue. We can see how proximal minimization produces the design $\mathbf{x}_1$ that is in $\mathcal{X}$ but closest to $\mathbf{v}$.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The ADMM update reduces to:[17]

$$\mathbf{x}_1^{(k+1)} = \arg\min_{\mathbf{x}_1 \in \mathcal{X}_1} \|\mathbf{x}_1 - \mathbf{x}_2^{(k)} + \mathbf{u}^{(k)}\|_2^2 \tag{11.62}$$

$$\mathbf{x}_2^{(k+1)} = \arg\min_{\mathbf{x}_2 \in \mathcal{X}_2} \|\mathbf{x}_1^{(k+1)} - \mathbf{x}_2 + \mathbf{u}^{(k)}\|_2^2 \tag{11.63}$$

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \mathbf{x}_1^{(k+1)} - \mathbf{x}_2^{(k+1)} \tag{11.64}$$

For this problem, the primal residual $\mathbf{r}^{(k)} = \mathbf{x}_1^{(k)} - \mathbf{x}_2^{(k)}$ is the difference between a point in $\mathcal{X}_1$ and a point in $\mathcal{X}_2$. Terminating the ADMM iterations when $\|\mathbf{r}^{(k)}\|_2$ falls below a threshold ϵ guarantees that $\mathcal{X}_1$ and $\mathcal{X}_2$ are no more than ϵ apart. This algorithm is demonstrated in figure 11.4.

[17] The update is obtained using equations (11.50) to (11.52). For example, the $\mathbf{x}_1$ update can be derived as follows:

$$\begin{aligned}
&\mathbf{x}_1^{(k+1)} \\
&= \arg\min_{\mathbf{x}_1} f_1(\mathbf{x}_1) + \frac{1}{2}\rho\left\|\mathbf{A}_1\mathbf{x}_1 - \mathbf{v}^{(k)}\right\|_2^2 \\
&= \arg\min_{\mathbf{x}_1 \in \mathcal{X}_1} \left\|\mathbf{x}_1 - \left(\mathbf{x}_2^{(k)} - \mathbf{u}^{(k)}\right)\right\|_2^2 \\
&= \arg\min_{\mathbf{x}_1 \in \mathcal{X}_1} \left\|\mathbf{x}_1 - \mathbf{x}_2^{(k)} + \mathbf{u}^{(k)}\right\|_2^2
\end{aligned}$$

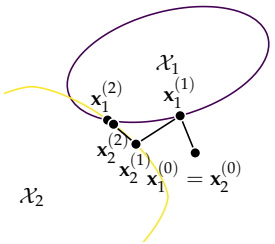


Figure 11.4. The first few iterations of the alternating projections method. The projections alternate between the two sets, drawing closer to the sets' intersection.

11.5.3 *Least Absolute Deviation*

Unconstrained least-squares problems can be formulated as quadratic programs, as covered in section 13.2. Least-squares problems assume that the distance metric being minimized is the L_2 norm. Using an L_1 norm instead results in a *least absolute deviation problem*, which is often preferred over least squares to provide robustness against large outliers:

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 \tag{11.65}$$

Such a problem can be written in ADMM form using $f_1(\mathbf{x}_1) = 0$ and letting $\mathbf{x}_2 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}$:

$$\begin{aligned}
&\underset{\mathbf{x}_1, \mathbf{x}_2}{\text{minimize}} && \|\mathbf{x}_2\|_1 \\
&\text{subject to} && \mathbf{A}\mathbf{x}_1 - \mathbf{x}_2 = \mathbf{b}
\end{aligned} \tag{11.66}$$

This decomposition neatly decouples the L_1 minimization problem from the affine constraint. If $\mathbf{A}^\top\mathbf{A}$ is invertible, the ADMM update is:[18]

$$\mathbf{x}_1' = \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top(\mathbf{b} + \mathbf{x}_2 - \mathbf{u}) \tag{11.67}$$

$$\mathbf{x}_2' = S_{\frac{1}{\rho}}\left(\mathbf{A}\mathbf{x}_1' - \mathbf{b} + \mathbf{u}\right) \tag{11.68}$$

$$\mathbf{u}' = \mathbf{u} + \mathbf{A}\mathbf{x}_1' - \mathbf{x}_2' - \mathbf{b} \tag{11.69}$$

[18] The update is obtained by using the scaled form updates, equations (11.50) to (11.52). For example, the $\mathbf{x}_1$ update is:

$$\begin{aligned}
&\mathbf{x}_1^{(k+1)} \\
&= \arg\min_{\mathbf{x}_1} f_1(\mathbf{x}_1) + \frac{1}{2}\rho\left\|\mathbf{A}\mathbf{x}_1 - \mathbf{v}^{(k)}\right\|_2^2 \\
&= \arg\min_{\mathbf{x}_1} \left\|\mathbf{A}\mathbf{x}_1 - \mathbf{v}^{(k)}\right\|_2^2 \\
&= \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\mathbf{v}^{(k)} \\
&= \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\left(-(-\mathbf{I})\mathbf{x}_2^{(k)} + \mathbf{b} - \mathbf{u}^{(k)}\right) \\
&= \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\left(\mathbf{b} + \mathbf{x}_2^{(k)} - \mathbf{u}^{(k)}\right)
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where the *soft thresholding operator* S_κ, shown in figure 11.5, is

$$S_\kappa(\mathbf{x})_i = \begin{cases} x_i + \kappa & \text{if } x_i < -\kappa \\ 0 & \text{if } |x_i| \leq \kappa \\ x_i - \kappa & \text{if } x_i > \kappa \end{cases} = \max(x_i - \kappa, 0) - \max(-x_i - \kappa, 0) \tag{11.70}$$

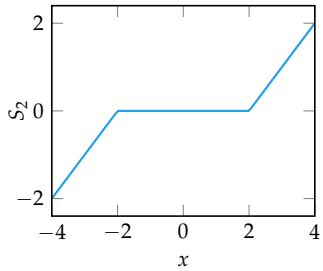


Figure 11.5. The soft thresholding operator for $\kappa = 2$ plotted in one dimension.

This problem can be solved very efficiently because we can compute the Cholesky factorization (appendix C.8.1) of $\mathbf{A}^\top\mathbf{A}$ once and re-use it to perform the $\mathbf{x}_1$-update in every iteration. An implementation is provided in algorithm 11.5.

```
function least_absolute_deviation(A, b, x1, x2; ρ=1, γ=2, ϵr=1e-3)
    u = zero(x2)
    r = fill(Inf, length(x2))
    F = factorize(A'*A)
    S(κ,x) = max.(x .- κ,0) - max.(-x .- κ,0)
    while norm(r) > ϵr
        x1′ = F \ A'*(b + x2 - u)
        x2′ = S(1/ρ, A*x1′ - b + u)
        u′ = A*x1′ - b + u - x2′
        r  = (A*x1′ - b) - x2′
        ρ *= γ
        x1, x2, u  = x1′, x2′, u′
    end
    return x1
end
```

Algorithm 11.5. ADMM applied to solve a least absolute deviation problem parameterized by `A` and `b`; initial design `x1` and `x2`; initial penalty scalar `ρ > 0` and penalty multiplier `γ > 1`; and convergence threshold `ϵr > 0`. The Julia method `factorize` will find an appropriate factorization for the given matrix, and due to type dispatching, solving $\mathbf{Fx} = \mathbf{b}$ through `F \ b` will use that factorization.

This method can be extended to other distance metrics, such as the *Huber function*:[19]

$$\underset{\mathbf{x}}{\text{minimize}} \quad \text{huber}(\mathbf{Ax} - \mathbf{b}) \tag{11.71}$$

where

$$\text{huber}(\mathbf{x}) = \sum_i \text{huber}(x_i) \tag{11.72}$$

and

$$\text{huber}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| \leq 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{11.73}$$

[19] Named for the Swiss statistician Peter Jost Huber (1934–) who introduced it in 1964. P. J. Huber, "Robust Estimation of a Location Parameter," *Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964. The Huber function is often parameterized such that the transition from quadratic to linear can be varied.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The Huber function is convex and often used for loss functions in machine learning. Being linear for large deviations and quadratic for small deviations makes it resilient to outliers while remaining continuously differentiable. The Huber function is shown in figure 11.6.

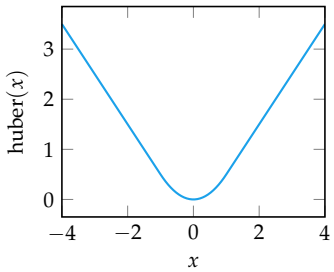


Figure 11.6. The Huber function plotted in one dimension.

The ADMM form of a minimum Huber deviation problem has the same structure:

$$\begin{aligned} \underset{\mathbf{x}_1, \mathbf{x}_2}{\text{minimize}} \quad & \text{huber}(\mathbf{x}_2) \\ \text{subject to} \quad & \mathbf{A}\mathbf{x}_1 - \mathbf{x}_2 = \mathbf{b} \end{aligned} \tag{11.74}$$

and has the same updates, except the $\mathbf{x}_2$-update is now:

$$\mathbf{x}_2' = \frac{\rho}{1+\rho}(\mathbf{A}\mathbf{x}_1' - \mathbf{b} + \mathbf{u}) + \frac{1}{1+\rho} S_{1+1/\rho}(\mathbf{A}\mathbf{x}_1' - \mathbf{b} + \mathbf{u}) \tag{11.75}$$

11.5.4 *Basis Pursuit*

Equality-constrained L_1 minimization problems, known as *basis pursuit* problems, have the form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x}\|_1 \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \end{aligned} \tag{11.76}$$

Such problems are solved to find sparse solutions to $\mathbf{A}\mathbf{x} = \mathbf{b}$ when the system of linear equations is underdetermined. Applying ADMM to this problem gives rise to the *split Bregman method*.[20] A basis pursuit problem can be written in ADMM form as:

$$\begin{aligned} \underset{\mathbf{x}_1, \mathbf{x}_2}{\text{minimize}} \quad & f_1(\mathbf{x}_1) + \|\mathbf{x}_2\|_1 \\ \text{subject to} \quad & \mathbf{x}_1 - \mathbf{x}_2 = \mathbf{0} \end{aligned} \tag{11.77}$$

where

$$f_1(\mathbf{x}_1) = \begin{cases} 0 & \text{if } \mathbf{A}\mathbf{x}_1 = \mathbf{b} \\ \infty & \text{otherwise} \end{cases} \tag{11.78}$$

[20] Named for the Soviet and Israeli mathematician Lev M. Bregman (1941–2023). L. Bregman, "The Relaxation Method of Finding the Common Point of Convex Sets and Its Application to the Solution of Problems in Convex Programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 3, pp. 200–217, 1967. T. Goldstein and S. Osher, "The Split Bregman Method for L_1-Regularized Problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 2, pp. 323–343, 2009.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

with the ADMM update:

$$\mathbf{x}_1' = \underset{\mathbf{x}_1 \mid \mathbf{A}\mathbf{x}_1 = \mathbf{b}}{\text{minimize}} \|(\mathbf{x}_2 - \mathbf{u}) - \mathbf{x}_1\|_2 \tag{11.79}$$

$$= \left(\mathbf{I} - \mathbf{A}^\top \left(\mathbf{A}\mathbf{A}^\top\right)^{-1} \mathbf{A}\right)(\mathbf{x}_2 - \mathbf{u}) + \mathbf{A}^\top \left(\mathbf{A}\mathbf{A}^\top\right)^{-1} \mathbf{b} \tag{11.80}$$

$$\mathbf{x}_2' = S_{\frac{1}{\rho}}\left(\mathbf{x}_1' + \mathbf{u}\right) \tag{11.81}$$

$$\mathbf{u}' = \mathbf{u} + \mathbf{x}_1' - \mathbf{x}_2' \tag{11.82}$$

The $\mathbf{x}_1$-update finds the $\mathbf{x}_1'$ closest to $\mathbf{x}_2 - \mathbf{u}$ that satisfies the equality constraint. In this case we can precompute most of the terms in the $\mathbf{x}_1$-update once and re-use them across iterations. We can again factorize $\mathbf{A}\mathbf{A}^\top$ once and use the factorization to compute the $\mathbf{x}_1$-update terms. An algorithm for solving basis pursuit problems with ADMM is given in algorithm 11.6.

```
function basis_pursuit(A, b, x1, x2; ρ=1, γ=2, ϵr=1e-3)
    u = zero(x2)
    r = fill(Inf, length(x2))

    F = factorize(A*A')
    C = I - A'*(F \ A)
    d = A'*(F \ b)
    S = (κ,x) -> max.(x .- κ,0) - max.(-x .- κ,0)

    while norm(r) > ϵr
        x1′ = C*(x2 - u) + d
        x2′ = S(1/ρ, x1′ + u)
        u′ = u + x1′ - x2′
        r  = x1′ - x2′
        ρ *= γ
        x1, x2, u  = x1′, x2′, u′
    end

    return x1
end
```

Algorithm 11.6. ADMM applied to solve a basis pursuit problem parameterized by `A` and `b`; initial designs `x1` and `x2`; initial penalty scalar `ρ > 0` and penalty multiplier `γ > 1`; and convergence threshold `ϵr > 0`.

11.5.5 *Lasso*

Linear regression is underdetermined when there are more features than data points. One common method for addressing both underdetermined problems and *overfitting*[21] the model to the data is to use *regularization*, where a penalty

[21] Overfitting is discussed in more detail in section 17.5.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

term is added to the objective function to prevent the solution from becoming too complex. If the penalty term is an L_1 norm, it is known as the *lasso*.[22] The L_1 term biases $\mathbf{x}^\star$ toward sparse solutions, where entries in $\mathbf{x}$ are encouraged to be zero, effectively downselecting the feature set. Lasso methods solve problems of the form:

$$\underset{\mathbf{x}}{\text{minimize}} \quad \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 + \lambda\|\mathbf{x}\|_1 \tag{11.83}$$

where $\lambda > 0$ scales the L_1 regularization term. The lasso method increases λ until a solution with the desired sparsity is found.

[22] The name comes from "least absolute shrinkage and selection operator." R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996.

An L_1-regularized linear regression problem can be written in ADMM form as:

$$\begin{aligned} \underset{\mathbf{x}_1,\mathbf{x}_2}{\text{minimize}} \quad & \frac{1}{2}\|\mathbf{A}\mathbf{x}_1 - \mathbf{b}\|_2^2 + \lambda\|\mathbf{x}_2\|_1 \\ \text{subject to} \quad & \mathbf{x}_1 - \mathbf{x}_2 = \mathbf{0} \end{aligned} \tag{11.84}$$

with the ADMM update:[23]

[23] The matrix $\mathbf{A}^\top\mathbf{A} + \rho\mathbf{I}$ for $\rho > 0$ is always invertible.

$$\mathbf{x}_1' = \left(\mathbf{A}^\top\mathbf{A} + \rho\mathbf{I}\right)^{-1}\left(\mathbf{A}^\top\mathbf{b} + \rho(\mathbf{x}_2 - \mathbf{u})\right) \tag{11.85}$$

$$\mathbf{x}_2' = S_{\frac{\lambda}{\rho}}\left(\mathbf{x}_1' + \mathbf{u}\right) \tag{11.86}$$

$$\mathbf{u}' = \mathbf{u} + \mathbf{x}_1' - \mathbf{x}_2' \tag{11.87}$$

Algorithm 11.7 solves L_1-regularized linear regression problems with ADMM.

```
function l1_regularized_linear_regression(
    A, b, λ, x1, x2; ρ=1, γ=2, ϵr=1e-3)

    u = zero(x2)
    r = fill(Inf, length(x2))
    S = (κ,x) -> max.(x .- κ,0) - max.(-x .- κ,0)
    while norm(r) > ϵr
        x1′ = (A'A + ρ*I) \ (A'b + ρ*(x2 - u))
        x2′ = S(λ/ρ, x1′ + u)
        u′ = u + x1′ - x2′
        r  = x1′ - x2′
        ρ *= γ
        x1, x2, u = x1′, x2′, u′
    end
    return x1
end
```

Algorithm 11.7. ADMM applied to solve an L_1-regularized linear regression problem, central to lasso, parameterized by `A`, `b`, and `λ`; initial design `x1` and `x2`; initial penalty scalar `ρ > 0` and penalty multiplier `γ > 1`; and convergence threshold `ϵr > 0`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

11.6 Distributed Methods

ADMM is amenable to distributed computation across multiple processors to improve efficiency. Modern problems in fields as diverse as artificial intelligence, computational biology, and logistics often operate on immense, typically high-dimensional, datasets. Many of these problems only become tractable when they can be processed in a distributed manner. The ADMM form can naturally be parallelized, making solving large problems tractable.

11.6.1 Consensus

One form of *consensus* problem consists of a sum of k convex objectives:[24]

$$\underset{\mathbf{x}}{\text{minimize}} \quad f_1(\mathbf{x}) + \cdots + f_k(\mathbf{x}) \tag{11.88}$$

[24] Terms can encode constraints by returning infinity when a constraint is violated.

This problem can be written in ADMM form by producing k separate copies of $\mathbf{x}_1$ and a single $\mathbf{x}_2$ vector:

$$\begin{aligned} &\underset{\mathbf{x}_1^{(1:k)}, \mathbf{x}_2}{\text{minimize}} && \sum_{i=1}^{k} f_i(\mathbf{x}_1^{(i)}) \\ &\text{subject to} && \mathbf{x}_1^{(i)} = \mathbf{x}_2 \qquad \text{for } i \text{ in } 1:k \end{aligned} \tag{11.89}$$

The copies of $\mathbf{x}_1$ all represent the same design variable, which all must necessarily match each other when solved. At the beginning, when ρ is small, each $\mathbf{x}_1^{(i)}$ will roughly independently minimize its own objective component. Consensus between the $\mathbf{x}_1$'s will ultimately be reached as ρ is increased, and all terms are penalized toward $\mathbf{x}_2$.

The ADMM update for a basic consensus problem is:[25]

[25] The multipliers sum to zero after the first iteration, making $\mathbf{x}_2'$ exactly the mean of the designs.

$$\left(\mathbf{x}_1^{(i)}\right)' = \underset{\mathbf{x}_1^{(i)}}{\arg\min}\, f_i(\mathbf{x}_1^{(i)}) + \left(\boldsymbol{\lambda}^{(i)}\right)^\top \left(\mathbf{x}_1^{(i)} - \mathbf{x}_2\right) + \frac{1}{2}\rho \left\| \mathbf{x}_1^{(i)} - \mathbf{x}_2 \right\|_2^2 \tag{11.90}$$

$$\mathbf{x}_2' = \frac{1}{k}\sum_{i=1}^{k}\left(\left(\mathbf{x}_1^{(i)}\right)' + \frac{1}{\rho}\boldsymbol{\lambda}^{(i)}\right) \quad \text{consensus step} \tag{11.91}$$

$$\left(\boldsymbol{\lambda}^{(i)}\right)' = \boldsymbol{\lambda}^{(i)} + \rho\left((\mathbf{x}_1^{(i)})' - \mathbf{x}_2'\right) \tag{11.92}$$

The primal and dual residuals are:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\mathbf{r} = \begin{bmatrix} \left(\mathbf{x}_1^{(1)}\right)' - \frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)' \\ \vdots \\ \left(\mathbf{x}_1^{(k)}\right)' - \frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)' \end{bmatrix} \qquad \mathbf{s} = -\rho \begin{bmatrix} \left(\frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)'\right) - \left(\frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)\right) \\ \vdots \\ \left(\frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)'\right) - \left(\frac{1}{k}\sum_{i=1}^{k}\left(\mathbf{x}_1^{(i)}\right)\right) \end{bmatrix} \tag{11.93}$$

The $\mathbf{x}_1^{(i)}$ updates can be parallelized across processes on a single machine, but can also be parallelized across multiple machines. Synchronization across processes or machines only requires that the inputs $\mathbf{x}_1^{(i)}$, $\boldsymbol{\lambda}^{(i)}$, $\mathbf{x}_2$, and ρ be passed back and forth. The implementations of each f_i, which may be large or contain sensitive information, need not be shared across update processes or the central consensus algorithm. As such, consensus approaches may not purely be motivated by speed.

An implementation is given in algorithm 11.8. The algorithm is demonstrated on a simple problem in figure 11.7.

11.6.2 *General Consensus Optimization*

We now consider the case where the local vectors $\mathbf{x}_1^{(i)}$ need not contain full copies of $\mathbf{x}_2$. In many cases, individual $\mathbf{x}_1^{(i)}$-updates need only consider a subset of the full set of design variables.

In *general consensus optimization*, the objective continues to be separable with $f(\mathbf{x}) = f_1(\mathbf{x}^{(1)}) + \cdots + f_k(\mathbf{x}^{(k)})$, but the local vectors may have different lengths. Each component of each local vector $\mathbf{x}_1^{(i)}$ corresponds to a component of the global variable $\mathbf{x}_2$. More than one $\mathbf{x}_1^{(i)}$ component may correspond to the same component of $\mathbf{x}_2$. When consensus is achieved, each component in $\mathbf{x}_1^{(i)}$ will equal its corresponding component in $\mathbf{x}_2$. We let $\mathbf{x}_2^{(i)}$ denote the vector of $\mathbf{x}_2$ components associated with the local design $\mathbf{x}_1^{(i)}$ such that $\mathbf{x}_1^{(i)} = \mathbf{x}_2^{(i)}$ at convergence.[26]

[26] This is merely convenient notation, and does not introduce additional variables.

We can formulate this problem as ADMM:

$$\begin{aligned} \underset{\mathbf{x}_1^{(1:k)}, \mathbf{x}_2}{\text{minimize}} \quad & \sum_{i=1}^{k} f_i(\mathbf{x}_1^{(i)}) \\ \text{subject to} \quad & \mathbf{x}_1^{(i)} = \mathbf{x}_2^{(i)} \qquad \text{for } i \text{ in } 1:k \end{aligned} \tag{11.94}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
using Base.Threads
function consensus(fs, x1s, x2; ρ=1, γ=2, ϵr=1e-3, ϵs=1e-3)
    k = length(fs)
    λs = [zeros(size(x2)) for i in 1:k]
    x1s′ = [zeros(size(x2)) for i in 1:k]

    norm_r = Inf
    norm_s = Inf
    x1_bar = mean(x1s)

    while norm_r > ϵr || norm_s > ϵs
        @threads for i in 1:k
            f, λ = fs[i], λs[i]
            obj = x1->f(x1) + λ⋅(x1-x2) + ρ/2*(x1-x2)⋅(x1-x2)
            x1s′[i] = minimize(obj, x1s[i])
        end

        x1_bar′ = mean(x1s′)
        x2′ = x1_bar′ + mean(λs)./ρ

        @threads for i in 1:k
            λs[i] += ρ*(x1s′[i] - x2′)
        end

        norm_r = sqrt(sum(norm(x1s′[i]-x1_bar′,2)^2 for i in 1:k))
        norm_s = ρ * sqrt(k)*norm(x1_bar′ - x1_bar, 2)
        ρ *= γ

        x1s[:], x2, x1_bar  = x1s′, x2′, x1_bar′
    end

    return x2
end
```

Algorithm 11.8. Consensus ADMM for solving a convex optimization problem with additive costs `fs`. The algorithm additionally receives initial designs `x1s` and `x2`; initial penalty scalar `ρ > 0` and penalty multiplier `γ > 1`; and convergence thresholds `ϵr` and `ϵs`.

This method executes its updates in parallel across available threads using the `@threads` macro. Julia can be started with multiple threads, for example with `julia -t 4`, or with the environment variable `JULIA_NUM_THREADS`.

For an overview of how to implement ADMM in distributed computing environments, see chapter 10 of S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

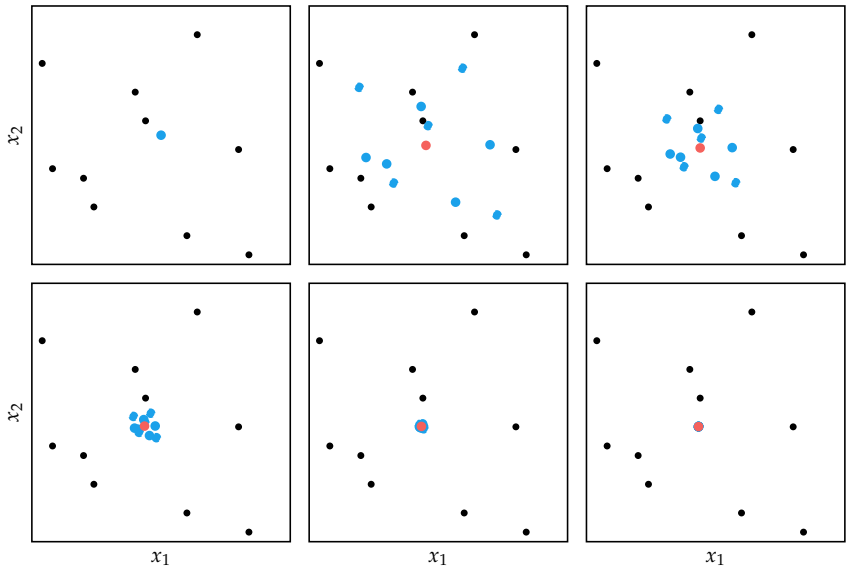


Figure 11.7. ADMM being used to solve a consensus problem in which we find a point that minimizes the sum of the squared L_2 distances to a set of points (black). The algorithm proceeds left to right, and top to bottom. The analytical solution will naturally be the mean of the points.

The designs (blue) and $\mathbf{x}_2$ (red) are initialized to the same value. There is a separate design for each target point, with each design's objective minimizing the squared distance to its target point. In the first iteration, each design moves most of the way toward its target. In subsequent iterations, as ρ increases, the designs converge to $\mathbf{x}_2$, which resides at the mean.

The ADMM update is:

$$\left(\mathbf{x}_1^{(i)}\right)' = \arg\min_{\mathbf{x}_1^{(i)}} f_i(\mathbf{x}_1^{(i)}) + \left(\boldsymbol{\lambda}^{(i)}\right)^\top \left(\mathbf{x}_1^{(i)} - \mathbf{x}_2^{(i)}\right) + \frac{1}{2}\rho \left\|\mathbf{x}_1^{(i)} - \mathbf{x}_2^{(i)}\right\|_2^2 \tag{11.95}$$

$$(\mathbf{x}_2)_g' = \frac{1}{n_g} \sum_{i=1}^{k} \left(\sum_j \left(\mathbf{x}_1^{(i)}\right)'_j \text{ where component } j \text{ corresponds to } (\mathbf{x}_2)_g' \right) \tag{11.96}$$

$$\text{for } g \text{ in } 1 : |\mathbf{x}_2| \tag{11.97}$$

$$\left(\boldsymbol{\lambda}^{(i)}\right)' = \boldsymbol{\lambda}^{(i)} + \rho\left((\mathbf{x}_1^{(i)})' - \left(\mathbf{x}_2^{(i)}\right)'\right) \tag{11.98}$$

where n_g is the number of components across all $\mathbf{x}_1$ designs that correspond to the gth component of $\mathbf{x}_2$. The $\mathbf{x}_1$ and λ updates can again be executed in parallel.

The $\mathbf{x}_2$-update continues to average over the local vectors, but because each local vector only contains certain components, it will only average the entries that correspond to each global component. Example 11.7 shows how a problem can be formulated as a global consensus problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 11.7. A bakery placement problem formulated as a general consensus problem. A solution is plotted below.

Suppose we are trying to place 3 bakeries in order to best serve 6 establishments:

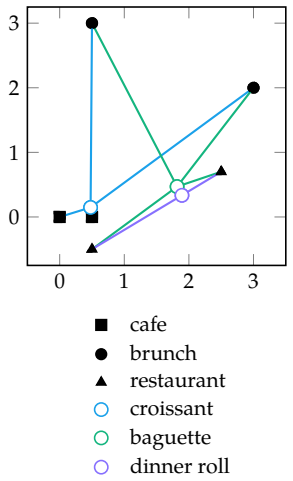
$$\begin{aligned}\text{two cafes at } \mathbf{e}^{(1)} &= [0.0, 0.0] \text{ and } \mathbf{e}^{(2)} = [0.5, 0.0]\\ \text{two brunch spots at } \mathbf{e}^{(3)} &= [3.0, 2.0] \text{ and } \mathbf{e}^{(4)} = [0.5, 3.0]\\ \text{two restaurants at } \mathbf{e}^{(5)} &= [0.5, -0.5] \text{ and } \mathbf{e}^{(6)} = [2.5, 0.7]\end{aligned}$$

One bakery, at $\mathbf{b}^{(1)}$, produces croissants, which are needed at cafes and brunch spots. Another bakery, at $\mathbf{b}^{(2)}$, produces baguettes, which are needed at brunch spots and restaurants. The third bakery, at $\mathbf{b}^{(3)}$, produces dinner rolls, which are only needed at restaurants. We want to minimize the distance from each establishment to its farthest compatible bakery:

$$\begin{aligned}\underset{\mathbf{b}^{(1:3)}}{\text{minimize}} \quad & \|\mathbf{e}^{(1)} - \mathbf{b}^{(1)}\| + \|\mathbf{e}^{(2)} - \mathbf{b}^{(1)}\| + \\ & \max(\|\mathbf{e}^{(3)} - \mathbf{b}^{(1)}\|, \|\mathbf{e}^{(3)} - \mathbf{b}^{(2)}\|) + \\ & \max(\|\mathbf{e}^{(4)} - \mathbf{b}^{(1)}\|, \|\mathbf{e}^{(4)} - \mathbf{b}^{(2)}\|) + \\ & \max(\|\mathbf{e}^{(5)} - \mathbf{b}^{(2)}\|, \|\mathbf{e}^{(5)} - \mathbf{b}^{(3)}\|) + \\ & \max(\|\mathbf{e}^{(6)} - \mathbf{b}^{(2)}\|, \|\mathbf{e}^{(6)} - \mathbf{b}^{(3)}\|)\end{aligned}$$

We can formulate this problem as a general consensus optimization problem with $\mathbf{x}_2 = [b_1^{(1)}, b_2^{(1)}, b_1^{(2)}, b_2^{(2)}, b_1^{(3)}, b_2^{(3)}]$:

$$\begin{aligned}\underset{\mathbf{x}_1^{(1:6)}, \mathbf{x}_2}{\text{minimize}} \quad & \|\mathbf{e}^{(1)} - \mathbf{x}_1^{(1)}\| + \|\mathbf{e}^{(2)} - \mathbf{x}_1^{(2)}\| + \\ & \max(\|\mathbf{e}^{(3)} - (\mathbf{x}_1^{(3)})_{1:2}\|, \|\mathbf{e}^{(3)} - (\mathbf{x}_1^{(3)})_{3:4}\|) + \\ & \max(\|\mathbf{e}^{(4)} - (\mathbf{x}_1^{(4)})_{1:2}\|, \|\mathbf{e}^{(4)} - (\mathbf{x}_1^{(4)})_{3:4}\|) + \\ & \max(\|\mathbf{e}^{(5)} - (\mathbf{x}_1^{(5)})_{1:2}\|, \|\mathbf{e}^{(5)} - (\mathbf{x}_1^{(5)})_{3:4}\|) + \\ & \max(\|\mathbf{e}^{(6)} - (\mathbf{x}_1^{(6)})_{1:2}\|, \|\mathbf{e}^{(6)} - (\mathbf{x}_1^{(6)})_{3:4}\|) \\ \text{subject to} \quad & \mathbf{x}_1^{(1)} = (\mathbf{x}_2)_{1:2} \quad \mathbf{x}_1^{(2)} = (\mathbf{x}_2)_{1:2} \quad \mathbf{x}_1^{(3)} = (\mathbf{x}_2)_{1:4} \\ & \mathbf{x}_1^{(4)} = (\mathbf{x}_2)_{1:4} \quad \mathbf{x}_1^{(5)} = (\mathbf{x}_2)_{3:6} \quad \mathbf{x}_1^{(6)} = (\mathbf{x}_2)_{3:6}\end{aligned}$$



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

11.7 Summary

- A constrained optimization problem has a dual problem formulation that can be easier to solve and whose solution is a lower bound of the solution to the original problem.
- The primal-dual method for inequality constrained problems directly seeks to satisfy the KKT conditions by optimizing the primal and dual variables simultaneously.
- Dual ascent is a technique for alternating between optimizing the primal and dual variables.
- The Alternating Direction Method of Multipliers is an extension of the method of multipliers and dual ascent for solving convex problems in a special canonical form.
- ADMM can be applied to any constrained optimization problem.
- ADMM proceeds by optimizing two objectives independently, but eventually reaches agreement by penalizing the separate design vectors toward one another over time.
- Multiple applications for ADMM were presented, giving rise to efficient algorithms simply by applying the ADMM machinery.
- Consensus algorithms can allow for parallelizing ADMM and running it in a decentralized manner.

11.8 Exercises

Exercise 11.1. Show how complementary slackness, equation (10.29), must hold for an optimal $(\mathbf{x}^*, \boldsymbol{\mu}^*, \boldsymbol{\lambda}^*)$ in a problem with a duality gap of zero.

Solution: A problem with a duality gap of zero will have equal primal and dual values:

$$p^* = f(\mathbf{x}^*) = \mathcal{D}(\boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = d^*$$

We know that the dual function is:

$$\mathcal{D}(\boldsymbol{\mu}^*, \boldsymbol{\lambda}^*) = \underset{\mathbf{x}}{\text{minimize}} \left(f(\mathbf{x}) + \sum_i \mu_i^* g_i(\mathbf{x}) + \sum_j \lambda_j^* h_j(\mathbf{x}) \right)$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The dual function is a lower bound on the Lagrangian:

$$\underset{\mathbf{x}}{\text{minimize}} \left(f(\mathbf{x}) + \sum_i \mu_i^* g_i(\mathbf{x}) + \sum_j \lambda_j^* h_j(\mathbf{x}) \right) \leq f(\mathbf{x}^*) + \sum_i \mu_i^* g_i(\mathbf{x}^*) + \sum_j \lambda_j^* h_j(\mathbf{x}^*)$$

Since we have zero duality gap, we have:

$$f(\mathbf{x}^*) + \sum_i \mu_i^* g_i(\mathbf{x}^*) + \sum_j \lambda_j^* h_j(\mathbf{x}^*) = f(\mathbf{x}^*)$$

and since each $h_j(\mathbf{x}^*) = 0$ and each $\mu_i^* g_i(\mathbf{x}^*) \leq 0$ at a solution, we conclude that complementary slackness must hold:

$$\mu_i^* g_i(\mathbf{x}^*) = 0 \text{ for all } i$$

Exercise 11.2. Is the dual problem for a dual problem the same as the primal problem?

Solution: Suppose we have a primal problem with value p^* and its dual with value d^*, and then we construct the dual of the dual problem with value dd^*. From the max-min inequality, we know that $d^* \leq p^*$ and that $dd^* \leq d^*$, so $dd^* \leq p^*$. As such, if there is a duality gap, the dual of a dual will not be equal to the primal problem. Hence, the dual of a dual is not necessarily the same as the primal problem.

Exercise 11.3. Consider the $\mathbf{x}_1$-update for an ADMM problem with a quadratic objective:

$$f_1(\mathbf{x}_1) = \frac{1}{2}\mathbf{x}_1{}^\top \mathbf{Q}\mathbf{x}_1 + \mathbf{q}^\top \mathbf{x}_1 + q$$

with positive semidefinite $\mathbf{Q}$. What is the update's analytic solution?

Solution: The update can be written:

$$\arg\min_{\mathbf{x}_1} \frac{1}{2}\mathbf{x}_1{}^\top \mathbf{Q}\mathbf{x}_1 + \mathbf{q}^\top \mathbf{x}_1 + q + \boldsymbol{\lambda}^\top \mathbf{x}_1 + \frac{1}{2}\rho \|\mathbf{A}_1\mathbf{x}_1 + \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}\|^2$$

We substitute $\mathbf{v} = \mathbf{q} + \boldsymbol{\lambda}$ and $\mathbf{w} = \mathbf{A}_2\mathbf{x}_2 - \mathbf{b}$. The $\mathbf{x}_1$-update becomes:

$$\begin{aligned}
&\arg\min_{\mathbf{x}_1} \frac{1}{2}\mathbf{x}_1{}^\top \mathbf{Q}\mathbf{x}_1 + \mathbf{v}^\top \mathbf{x}_1 + q + \frac{1}{2}\rho \|\mathbf{A}_1\mathbf{x}_1 + \mathbf{w}\|^2 \\
&\arg\min_{\mathbf{x}_1} \frac{1}{2}\mathbf{x}_1{}^\top \mathbf{Q}\mathbf{x}_1 + \mathbf{v}^\top \mathbf{x}_1 + \frac{1}{2}\rho \mathbf{x}_1^\top \mathbf{A}_1^\top \mathbf{A}_1\mathbf{x}_1 - \rho \mathbf{w}^\top \mathbf{A}_1\mathbf{x}_1 + \frac{1}{2}\rho \mathbf{w}^\top \mathbf{w} \\
&\arg\min_{\mathbf{x}_1} \frac{1}{2}\mathbf{x}_1{}^\top \left(\mathbf{Q} + \rho \mathbf{A}_1^\top \mathbf{A}_1\right)\mathbf{x}_1 + \left(\mathbf{v} - \rho \mathbf{A}_1^\top \mathbf{w}\right)^\top \mathbf{x}_1
\end{aligned}$$

The updated design $\mathbf{x}_1'$ can thus be found by solving:

$$\left(\mathbf{Q} + \rho \mathbf{A}_1^\top \mathbf{A}_1\right)\mathbf{x}_1' = \left(\rho \mathbf{A}_1^\top (\mathbf{A}_2\mathbf{x}_2 - \mathbf{b}) - \mathbf{q} - \boldsymbol{\lambda}\right)$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Running ADMM on this problem thus requires repeatedly solving similar systems of linear equations $\mathbf{Q}_\rho \mathbf{x}_1' = \mathbf{q}_\rho$. Such a linear system is typically solved by computing the Cholesky decomposition $\mathbf{L}\mathbf{L}^\top = \mathbf{Q}_\rho$ and then efficiently solving $\mathbf{L}\mathbf{L}^\top \mathbf{x}_1' = \mathbf{q}_\rho$. We can save computation by limiting how often ρ is updated, thereby allowing the factorization for $\mathbf{Q}_\rho$ to be reused.[27]

[27] For more examples of how structure can be exploited to improve efficiency, see section 4.2 of S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

Exercise 11.4. Show that the dual residual, equation (11.39), measures how close the design is to satisfying stationarity in $\mathbf{x}_1$.

Solution: The $\mathbf{x}_1$ update minimizes $\mathcal{L}_\rho\left(\mathbf{x}_1, \mathbf{x}_2^{(k)}, \boldsymbol{\lambda}^{(k)}\right)$. As such, the resulting $\mathbf{x}_1^{(k+1)}$ satisfies

$$
\begin{aligned}
\mathbf{0} &= \nabla_{\mathbf{x}_1} \mathcal{L}_\rho\left(\mathbf{x}_1^{(k+1)}, \mathbf{x}_2^{(k)}, \boldsymbol{\lambda}^{(k)}\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \boldsymbol{\lambda}^{(k)} + \rho \mathbf{A}_1^\top \left(\mathbf{A}_1 \mathbf{x}_1^{(k+1)} + \mathbf{A}_2 \mathbf{x}_2^{(k)} - \mathbf{b}\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \left(\boldsymbol{\lambda}^{(k)} + \rho \mathbf{r}^{(k+1)} - \rho \mathbf{r}^{(k+1)} + \rho\left(\mathbf{A}_1 \mathbf{x}_1^{(k+1)} + \mathbf{A}_2 \mathbf{x}_2^{(k)} - \mathbf{b}\right)\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \left(\boldsymbol{\lambda}^{(k)} + \rho \mathbf{r}^{(k+1)} - \rho\left(\mathbf{A}_1 \mathbf{x}_1^{(k+1)} + \mathbf{A}_2 \mathbf{x}_2^{(k+1)} - \mathbf{b}\right) + \rho\left(\mathbf{A}_1 \mathbf{x}_1^{(k+1)} + \mathbf{A}_2 \mathbf{x}_2^{(k)} - \mathbf{b}\right)\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \left(\boldsymbol{\lambda}^{(k)} + \rho \mathbf{r}^{(k+1)} + \rho \mathbf{A}_2\left(\mathbf{x}_2^{(k)} - \mathbf{x}_2^{(k+1)}\right)\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \left(\boldsymbol{\lambda}^{(k+1)} + \rho \mathbf{A}_2\left(\mathbf{x}_2^{(k)} - \mathbf{x}_2^{(k+1)}\right)\right) \\
&= \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \boldsymbol{\lambda}^{(k+1)} + \rho \mathbf{A}_1^\top \mathbf{A}_2\left(\mathbf{x}_2^{(k)} - \mathbf{x}_2^{(k+1)}\right)
\end{aligned}
$$

Solving and rearranging produces:

$$\rho \mathbf{A}_1^\top \mathbf{A}_2\left(\mathbf{x}_2^{(k+1)} - \mathbf{x}_2^{(k)}\right) = \nabla_{\mathbf{x}_1} f_1(\mathbf{x}_1^{(k+1)}) + \mathbf{A}_1^\top \boldsymbol{\lambda}^{(k+1)}$$

The left hand side of the above equation is the dual residual, and the right hand side is the gradient of the unmodified Lagrangian, which must be zero when stationarity is satisfied. This means the dual residual measures how close the design is to satisfying stationarity in $\mathbf{x}_1$ for the original problem equation (11.30).

Exercise 11.5. Prior to Dykstra's method for alternating projections (section 11.5.2), John von Neumann used a simpler algorithm for finding a point in the intersection of two affine sets that does not use the dual variable $\mathbf{u}$:[28]

[28] J. Von Neumann, "On Rings of Operators. Reduction Theory," *Annals of Mathematics*, pp. 401–485, 1949.

$$
\begin{aligned}
\mathbf{x}_1^{(k+1)} &= \operatorname*{arg\,min}_{\mathbf{x}_1 \in \mathcal{X}_1} \|\mathbf{x}_1 - \mathbf{x}_2^{(k)}\| \\
\mathbf{x}_2^{(k+1)} &= \operatorname*{arg\,min}_{\mathbf{x}_2 \in \mathcal{X}_2} \|\mathbf{x}_1^{(k+1)} - \mathbf{x}_2\|
\end{aligned}
$$

where $\mathcal{X}_1 = \{\mathbf{x} \mid \mathbf{C}_1 \mathbf{x} \leq \mathbf{d}_1\}$ and $\mathcal{X}_2 = \{\mathbf{x} \mid \mathbf{C}_2 \mathbf{x} \leq \mathbf{d}_2\}$.

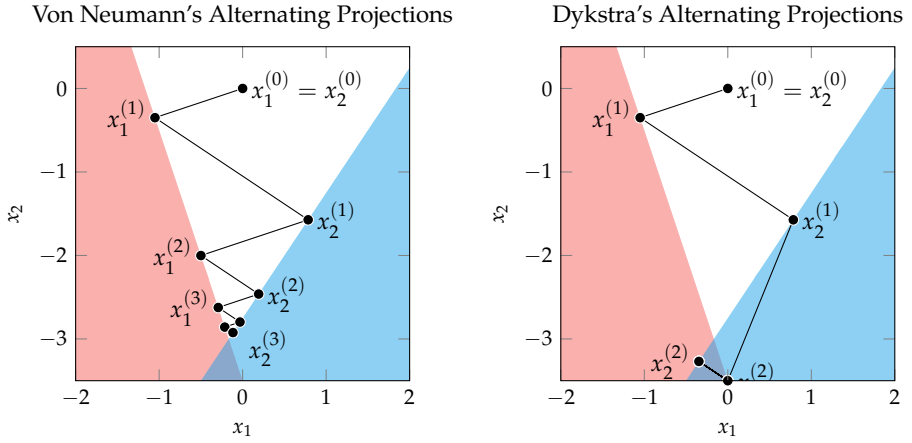
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Run both Dykstra's and von Neumann's alternating projections algorithms on the sets formed by

$$\mathbf{C}_1 = \begin{bmatrix} 3.0 & 1.0 \\ 0.5 & -1.0 \end{bmatrix} \qquad \mathbf{d}_1 = \begin{bmatrix} -3.5 \\ 3.5 \end{bmatrix}$$
$$\mathbf{C}_2 = \begin{bmatrix} -1.5 & 1.0 \\ -0.9 & -1.0 \end{bmatrix} \qquad \mathbf{d}_2 = \begin{bmatrix} -2.75 \\ 3.95 \end{bmatrix}$$

from $\mathbf{x}_1^{(0)} = \mathbf{x}_2^{(0)} = \mathbf{0}$. Does the dual variable help accelerate convergence?

Solution: Each algorithm's progression is shown below:



Von Neumann's alternating projections method is slower to converge in narrow canyons, where the algorithm has to slowly zig-zag its way toward a feasible point. The ADMM approach, which is equivalent to Dykstra's alternating projections method, has the same first iteration, but the dual vector $\mathbf{u}$ acts as a momentum term that propels subsequent iterations.

Exercise 11.6. Show how the ADMM update for a consensus problem can be simplified, removing the $\mathbf{x}_2$-update entirely. As a hint, replace $(1/k)\sum_{i=1}^{k} \mathbf{x}_1^{(i)}$ with $\bar{\mathbf{x}}_1$, and show that the $\mathbf{x}_2$-update can be written as $\mathbf{x}_2' = \bar{\mathbf{x}}_1'$.

Solution: The $\mathbf{x}_2$-update for a consensus problem:

$$\mathbf{x}_2' = \frac{1}{k}\sum_{i=1}^{k}\left(\left(\mathbf{x}_1^{(i)}\right)' + \frac{1}{\rho}\boldsymbol{\lambda}^{(i)}\right)$$

can be simplified to:

$$\mathbf{x}_2' = \bar{\mathbf{x}}_1' + \frac{1}{\rho}\bar{\boldsymbol{\lambda}}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The average over the dual variables, $\bar{\boldsymbol{\lambda}}$, is updated according to:

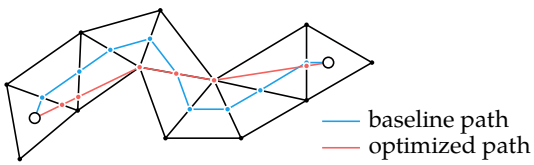
$$\begin{aligned}\bar{\boldsymbol{\lambda}}' &= \bar{\boldsymbol{\lambda}} + \rho(\bar{\mathbf{x}}_1' - \mathbf{x}_2') \\ &= \bar{\boldsymbol{\lambda}} + \rho\left(\bar{\mathbf{x}}_1' - \left(\bar{\mathbf{x}}_1' + \frac{1}{\rho}\bar{\boldsymbol{\lambda}}\right)\right) \\ &= \mathbf{0}\end{aligned}$$

As such, we can drop the second term from the $\mathbf{x}_2$-update and use $\mathbf{x}_2' = \bar{\mathbf{x}}_1'$, resulting in the ADMM update:

$$\left(\mathbf{x}_1^{(i)}\right)' = \arg\min_{\mathbf{x}_1}\left(f_i(\mathbf{x}_1^{(i)}) + \left(\boldsymbol{\lambda}^{(i)}\right)^\top\left(\mathbf{x}_1^{(i)} - \bar{\mathbf{x}}_1\right) + \frac{1}{2}\rho\left\|\mathbf{x}_1^{(i)} - \bar{\mathbf{x}}_1\right\|_2^2\right)$$

$$\left(\boldsymbol{\lambda}^{(i)}\right)' = \boldsymbol{\lambda}^{(i)} + \rho\left((\mathbf{x}_1^{(i)})' - \bar{\mathbf{x}}_1'\right)$$

Exercise 11.7. Suppose we have run a shortest-path algorithm on a constrained triangle mesh, and now have a path composed of tile traversals, using the center of each triangle edge. We now wish to locally optimize this path to find the shortest path that continues to pass through the same triangles as before.[29] How can this problem be framed using general consensus optimization?[30]

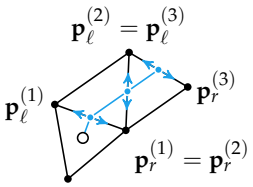


Solution: To formulate this problem to be solved using general consensus, we first need to formulate it as an optimization problem. We wish to minimize the total path length. Our path traverses through a series of predefined triangles. We are free, however, to change where in these triangles we traverse.

Let $\mathbf{p}^{(0)}$ be the starting vertex, $\mathbf{p}^{(n+1)}$ be the destination vertex, and $\mathbf{p}^{(i)}$ for i in $1:n$ be the n vertices at the triangle edges that our path must traverse. We can slide each $\mathbf{p}^{(i)}$ between its leftmost and rightmost value:

$$\mathbf{p}^{(i)} = \alpha_i \mathbf{p}_\ell^{(i)} + (1 - \alpha_i)\mathbf{p}_r^{(i)}$$

where each interpolant α_i lies in the unit bounds, $\alpha_i \in [0, 1]$.



[29] This local form of path optimization is sometimes referred to as an *elastic band* optimization. We can imagine the path being turned into a tightened elastic band that stretches and presses against its bounds, forming the shortest path.

[30] The overhead associated with parallelizing this problem is likely not worth it beyond its merits as an academic exercise, particularly because this problem can be formulated as a quadratic program and thus can be solved very efficiently. However, variations on this problem, such as using connecting segments that respect vehicle dynamic limits and motion constraints like maximum turning rates can quickly increase problem complexity, which can make general consensus optimization worthwhile.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The total path length is

$$\sum_{i=0}^{n} \|\mathbf{p}^{(i+1)} - \mathbf{p}^{(i)}\|_2$$

Putting this together, our optimization problem is

$$\begin{aligned} \underset{\boldsymbol{\alpha}_{1:n}}{\text{minimize}} \quad & \sum_{i=0}^{n} \left\| \mathbf{p}^{(i+1)} - \mathbf{p}^{(i)} \right\|_2 \\ \text{subject to} \quad & \mathbf{p}^{(i)} = \alpha_i \mathbf{p}_\ell^{(i)} + (1 - \alpha_i)\mathbf{p}_r^{(i)} \\ & \mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{1} \end{aligned}$$

This problem can be framed using general consensus. The objective function is already broken apart into components, with each component requiring two interpolants:

$$\begin{aligned} f_i(\alpha_i, \alpha_{i+1}) &= \left\| \mathbf{p}^{(i+1)} - \mathbf{p}^{(i)} \right\|_2 \\ &= \left\| \left(\alpha_{i+1} \mathbf{p}_\ell^{(i+1)} + (1 - \alpha_{i+1}) \mathbf{p}_r^{(i+1)} \right) - \left(\alpha_i \mathbf{p}_\ell^{(i)} + (1 - \alpha_i) \mathbf{p}_r^{(i)} \right) \right\|_2 \end{aligned}$$

The interpolants must all lie within the unit bounds. This can be enforced as an additional objective:

$$f_{n+1}(\boldsymbol{\alpha}) = \begin{cases} 0 & \text{if } \mathbf{0} \leq \boldsymbol{\alpha} \leq 1 \\ \infty & \text{otherwise} \end{cases}$$

However, it is often simpler to enforce the unit bounds in the $\mathbf{x}_1^{(i)}$-updates:

$$\left(\mathbf{x}_1^{(i)} \right)' = \underset{\mathbf{0} \leq \mathbf{x}_1 \leq 1}{\arg\min} \, f_i(\mathbf{x}_1^{(i)}) + \left(\boldsymbol{\lambda}^{(i)} \right)^\top \left(\mathbf{x}_1^{(i)} - \mathbf{x}_2^{(i)} \right) + \frac{1}{2} \rho \left\| \mathbf{x}_1^{(i)} - \mathbf{x}_2^{(i)} \right\|_2^2$$

In general consensus, we would maintain a global copy of the interpolants, $\mathbf{x}_2 = \boldsymbol{\alpha}$. Each component would then optimize a local vector $\mathbf{x}_1^{(i)}$ that only contains entries corresponding to relevant components.

Exercise 11.8. Consider a simplified model of economic trade in the United States with three commodities: artichokes, oranges, and potatoes. The states California, Florida, and Idaho each produce some quantity of these commodities every year:

$$\begin{aligned} \mathbf{p}_C &= [3.0, 1.5, 1.5] \\ \mathbf{p}_F &= [0.0, 3.0, 0.5] \\ \mathbf{p}_I &= [0.0, 0.0, 5.0] \end{aligned}$$

Each state can give commodities to or receive commodities from a central exchange. Let the amount of commodities received by a state be represented by $\mathbf{x}$. For example, $\mathbf{x}_C = [-0.6, 0.2, 0.3]$ would mean that California provides 0.6 units of artichokes to the exchange, but then receives 0.2 units of oranges and 0.3 units of potatoes. The exchange must enforce equilibrium, with the net inflow of every commodity matching its outflow:

$$\sum_{s \in (C,F,I)} \mathbf{x}_s = \mathbf{0}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Each state desires all commodities according to a sigmoid utility function

$$u(c) = \frac{2}{1 + e^{-c}} - 1$$

where $c = p + x$ is the amount of a particular commodity that the state ends up with. A state's total utility depends both on the amount of a particular commodity and the total amount of commodities:

$$U(\mathbf{c}) = u(c_{\text{artichokes}}) + u(c_{\text{oranges}}) + u(c_{\text{potatoes}}) + u(\|\mathbf{c}\|)$$

Express this problem in ADMM form, despite the fact that the utility function is nonconvex. What is its ADMM update, in scaled form?

Solution: Our optimization problem is

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & \sum_{s \in \mathcal{S}} U(\mathbf{p}_s + \mathbf{x}_s) \\ \text{subject to} \quad & \sum_{s \in \mathcal{S}} \mathbf{x}_s = \mathbf{0} \end{aligned}$$

where $\mathcal{S} = \{C, F, I\}$ is the set of states.

The objective function is nonconvex. Nevertheless, the corresponding ADMM form is

$$\begin{aligned} \underset{\mathbf{x},\, \mathbf{z}}{\text{minimize}} \quad & \left(\sum_{s \in \mathcal{S}} -U(\mathbf{p}_s + \mathbf{x}_s) \right) + (\sum_{s \in \mathcal{S}} \mathbf{z}_s \neq \mathbf{0}) \cdot \infty \\ \text{subject to} \quad & \mathbf{x} - \mathbf{z} = \mathbf{0} \end{aligned}$$

where we have used $\mathbf{x}$ and $\mathbf{z}$ instead of $\mathbf{x}_1$ and $\mathbf{x}_2$ for notational convenience.

The ADMM update for this problem in scaled form is thus

$$\begin{aligned} \mathbf{x}_s^{(k+1)} &= \arg\min_{\mathbf{x}} -U(\mathbf{p}_s + \mathbf{x}) + \frac{1}{2}\rho \sum_{s \in \mathcal{S}} \left\| \mathbf{x}_s - \mathbf{z}_s^{(k)} + \mathbf{u}_s^{(k)} \right\|_2^2 \\ \mathbf{z}^{(k+1)} &= \arg\min_{\mathbf{z} | \sum_{s \in \mathcal{S}} \mathbf{z}_s = \mathbf{0}} \frac{1}{2}\rho \sum_{s \in \mathcal{S}} \left\| \mathbf{x}_s^{(k+1)} - \mathbf{z}_s + \mathbf{u}_s^{(k)} \right\|_2^2 \\ \mathbf{u}_s^{(k+1)} &= \mathbf{u}_s^k + \mathbf{x}_s^{(k+1)} - \mathbf{z}_s^{(k+1)} \end{aligned}$$

Exercise 11.9. We can simplify the $\mathbf{z}$-update for the exchange problem in the previous exercise by noticing that solving the $\mathbf{z}$-update produces:

$$\mathbf{z}_s^{(k+1)} = \mathbf{x}_s^{(k+1)} + \mathbf{u}_s^{(k)} - \frac{1}{|\mathcal{S}|} \sum_{t \in \mathcal{S}} \left(\mathbf{x}_t^{(k+1)} + \mathbf{u}_t^{(k)} \right) = \mathbf{x}_s^{(k+1)} + \mathbf{u}_s^{(k)} - \bar{\mathbf{x}}^{(k+1)} - \bar{\mathbf{u}}^{(k)}$$

Make this substitution and derive a simplified ADMM update that neither requires a z-update nor separate $\mathbf{u}$ vectors for each state. Can any aspects of this problem be solved in parallel?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: Substituting the updated $\mathbf{z}$ vectors into the u-update produces:

$$\begin{aligned}\mathbf{u}_s^{(k+1)} &= \mathbf{u}_s^{(k)} + \mathbf{x}_s^{(k+1)} - \mathbf{z}_s^{(k+1)} \\ &= \mathbf{u}_s^{(k)} + \mathbf{x}_s^{(k+1)} - \left(\mathbf{x}_s^{(k+1)} + \mathbf{u}_s^{(k)} - \bar{\mathbf{x}}^{(k+1)} - \bar{\mathbf{u}}^{(k)}\right) \\ &= \bar{\mathbf{u}}^{(k)} + \bar{\mathbf{x}}^{(k+1)}\end{aligned}$$

from which we can see that the $\mathbf{u}_s$ values are all equal. We can thus use a single $\mathbf{u}$.

The final procedure for this exchange problem is:

$$\begin{aligned}\mathbf{x}_s^{(k+1)} &= \arg\min_{\mathbf{x}} -U(\mathbf{p}_s + \mathbf{x}) + \frac{1}{2}\rho\left\|\mathbf{x}_s - \mathbf{x}_s^{(k)} + \bar{\mathbf{x}}^{(k)} + \mathbf{u}^{(k)}\right\|_2^2 \\ \mathbf{u}^{(k+1)} &= \mathbf{u}^k + \bar{\mathbf{x}}^{(k+1)}\end{aligned}$$

for which the x-updates can all be carried out in parallel.

Exercise 11.10. Solve the exchange problem from the previous two exercises.

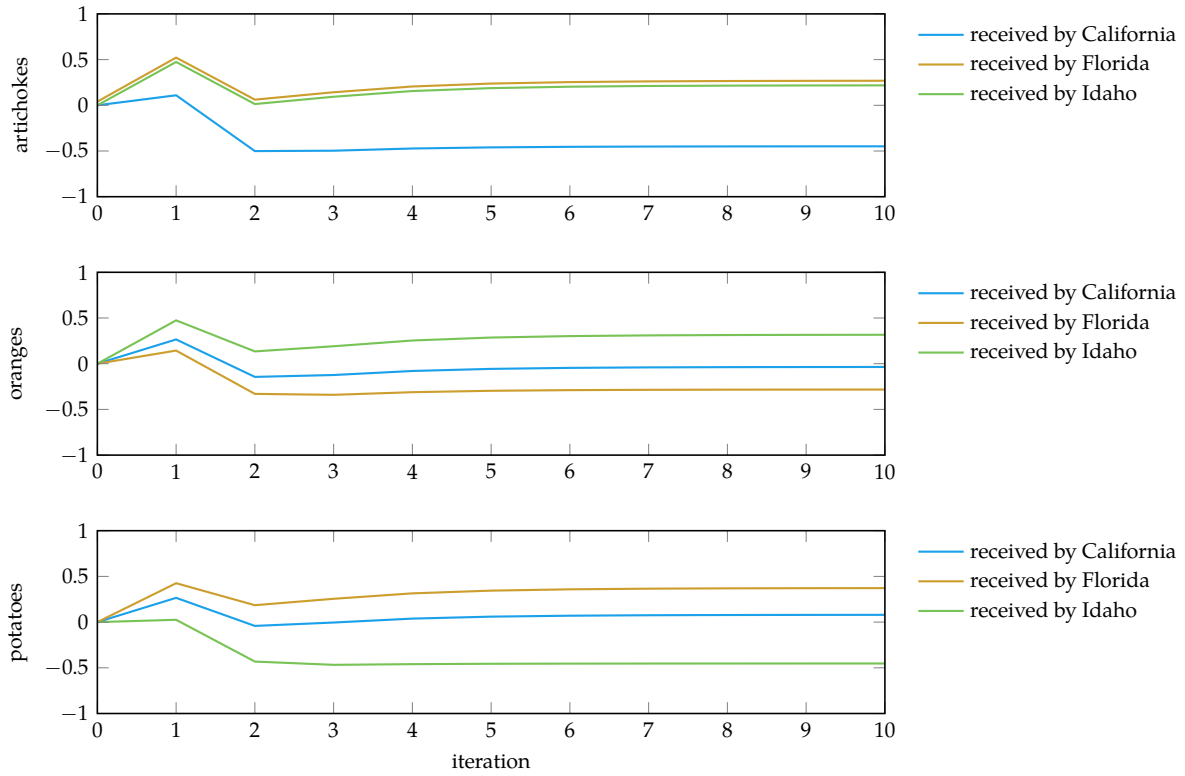
Solution: The objective function is not convex, but descent methods can still be used to solve the x-updates within the ADMM procedure.

Solving the problem yields:

$$\begin{aligned}\mathbf{x}_C &= [-0.448, -0.035, \quad 0.080] \\ \mathbf{x}_F &= [\quad 0.229, -0.282, \quad 0.373] \\ \mathbf{x}_I &= [\quad 0.219, \quad 0.317, -0.453]\end{aligned}$$

The goods received by each state as the algorithm progresses are presented below. We can see how each state primarily trades away the good that it has in excess in order to receive goods that it lacks from other states. The initial quantities do not sum to zero, but as the algorithm progresses and ρ increases, that constraint is increasingly enforced.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 11.11. Consider a problem of the form[31]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{g}(\mathbf{x}) \leq \mathbf{1} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{1} \\ & \mathbf{x} > \mathbf{0} \end{aligned}$$

where each equality constraint function has the form

$$h_i(\mathbf{x}) = c_i x_1^{a_{i1}} x_2^{a_{i2}} \cdots x_m^{a_{im}} = c_i \prod_j x_j^{a_{ij}}$$

for $c_i > 0$ and $a_{ij} \in \mathbb{R}$, called a *monomial*. The objective function and the inequality constraint functions are sums of any number of monomials.

[31] A problem in this form is called a *geometric program*. Modern interior-point algorithms (see algorithm 10.4) can solve a geometric program with 1,000 variables and 10,000 constraints in under a minute. S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A Tutorial on Geometric Programming," *Optimization and Engineering*, vol. 8, pp. 67–127, 2007.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Problems of this form can be converted into convex problems by taking the log of the objective function and all constraint functions and applying the change of variables $z_i = \log x_i$. Show that these changes produce a problem with a log-sum-exp objective, log-sum-exp inequality constraints, and affine equality constraints.

Solution: The log of a monomial is:

$$\log\left(c_i \prod_j x_j^{a_{ij}}\right) = \log c_i + \sum_j a_{ij} \log x_j$$

Applying the change of variables $z_j = \log x_j$ results in an affine constraint:

$$\log\left(c_i \prod_j x_j^{a_{ij}}\right) = 0 \quad \Rightarrow \quad \sum_j a_{ij} z_j = -\log c_i$$

The log of a sum of monomials is:

$$\log\left(\sum_i c_i \prod_j x_j^{a_{ij}}\right)$$

If we apply the same change of variables, rewritten as $\exp z_j = x_j$, we get a function in log-sum-exp form:

$$\log\left(\sum_i c_i \prod_j \left(\exp z_j\right)^{a_{ij}}\right) = \log\left(\sum_i \exp\left(\log c_i + \sum_j a_{ij} z_j\right)\right)$$

The resulting problem thus has such a log-sum-exp objective, log-sum-exp inequality constraints, and affine equality constraints.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

12 Linear Programming

Linear programming involves solving optimization problems with linear objective functions and linear constraints. Many problems are naturally described by linear programs, including problems from fields as diverse as transportation, communication networks, manufacturing, economics, and operations research. Many problems that are not naturally linear can often be approximated by linear programs. Although linear programs can be solved using the interior point methods from section 10.11, this chapter focuses on the simplex algorithm, which exploits the linear structure of the problem. Modern techniques and hardware can globally minimize problems with millions of variables and millions of constraints.[1]

[1] Several textbooks are dedicated entirely to linear programs, including R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, 4th ed. Springer, 2014. There are a variety of packages for solving linear programs, such as `Convex.jl` and `JuMP.jl`, both of which include interfaces to open-source and commercial solvers.

12.1 Problem Formulation

A linear programming problem, called a *linear program*, can be expressed in several forms. Each linear program consists of a linear objective function and a set of linear constraints:[2]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{w}_{\text{LE}}^{(i)\top} \mathbf{x} \leq b_i \quad \text{for all } i \\ & \mathbf{w}_{\text{EQ}}^{(j)\top} \mathbf{x} = b_j \quad \text{for all } j \end{aligned} \tag{12.1}$$

[2] As discussed in section 10.2, we can transform greater-than inequalities into less-than inequalities.

where i and j vary over finite sets of constraints. Such an optimization problem is given in example 12.1. Transforming real problems into this mathematical form is often nontrivial. This text focuses on the algorithms for obtaining solutions, but other texts discuss how to go about modeling real problems.[3] Several interesting conversions are given in example 12.2.

[3] See, for example, H. P. Williams, *Model Building in Mathematical Programming*, 5th ed. Wiley, 2013.

Example 12.1. An example linear program.

The following problem has a linear objective and linear constraints, making it a linear program.

$$\begin{aligned}
\underset{x_1,x_2,x_3}{\text{minimize}} \quad & 2x_1 - 3x_2 + 7x_3 \\
\text{subject to} \quad & 2x_1 + 3x_2 - 8x_3 \leq 5 \\
& 4x_1 + x_2 + 3x_3 \leq 9 \\
& x_1 - 5x_2 - 3x_3 \geq -4 \\
& x_1 + x_2 + 2x_3 = 1
\end{aligned}$$

Example 12.2. Common norm minimization problems that can be converted into linear programs.

Many problems can be converted into linear programs that have the same solution. Two examples are L_1 and L_∞ minimization problems:

$$\text{minimize} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 \qquad \text{minimize} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_\infty$$

The first problem is equivalent to solving

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{s}}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{s} \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{s} \\
& \mathbf{A}\mathbf{x} - \mathbf{b} \geq -\mathbf{s}
\end{aligned}$$

with the additional variables $\mathbf{s}$.

The second problem is equivalent to solving

$$\begin{aligned}
\underset{\mathbf{x},t}{\text{minimize}} \quad & t \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} - \mathbf{b} \leq t\mathbf{1} \\
& \mathbf{A}\mathbf{x} - \mathbf{b} \geq -t\mathbf{1}
\end{aligned}$$

with the additional variable t.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

12.1.1 General Form

We can write linear programs more compactly using matrices and arrive at the *general form*:[4]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}_{\text{LE}}\mathbf{x} \leq \mathbf{b}_{\text{LE}} \\ & \mathbf{A}_{\text{EQ}}\mathbf{x} = \mathbf{b}_{\text{EQ}} \end{aligned} \tag{12.2}$$

[4] Here, each constraint is element-wise. For example, in writing

$$\mathbf{a} \leq \mathbf{b},$$

we mean $a_i \leq b_i$ for all i.

12.1.2 Standard Form

The general linear program given in equation (12.2) can be converted into *standard form* where all constraints are less-than inequalities and the design variables are nonnegative

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{12.3}$$

Equality constraints are split in two:

$$\mathbf{A}_{\text{EQ}}\mathbf{x} = \mathbf{b}_{\text{EQ}} \rightarrow \begin{cases} \mathbf{A}_{\text{EQ}}\mathbf{x} \leq \mathbf{b}_{\text{EQ}} \\ -\mathbf{A}_{\text{EQ}}\mathbf{x} \leq -\mathbf{b}_{\text{EQ}} \end{cases} \tag{12.4}$$

We must ensure that all $\mathbf{x}$ entries are nonnegative as well. Suppose we start with a linear program where $\mathbf{x}$ is not constrained to be nonnegative:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \end{aligned} \tag{12.5}$$

We replace $\mathbf{x}$ with $\mathbf{x}^+ - \mathbf{x}^-$ and constrain $\mathbf{x}^+ \geq \mathbf{0}$ and $\mathbf{x}^- \geq \mathbf{0}$:

$$\begin{aligned} \underset{\mathbf{x}^+, \mathbf{x}^-}{\text{minimize}} \quad & \begin{bmatrix} \mathbf{c}^\top & -\mathbf{c}^\top \end{bmatrix} \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} \mathbf{A} & -\mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \leq \mathbf{b} \\ & \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \geq \mathbf{0} \end{aligned} \tag{12.6}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The linear objective function $\mathbf{c}^\top \mathbf{x}$ forms a flat ramp. The function increases in the direction of $\mathbf{c}$, and, as a result, all contour lines are perpendicular to $\mathbf{c}$ and parallel to one another as shown in figure 12.1.

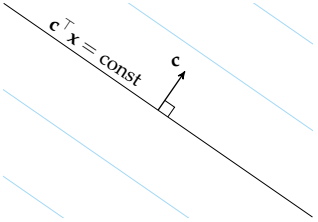


Figure 12.1. The contours of a linear objective function $\mathbf{c}^\top \mathbf{x}$, which increase in the direction of $\mathbf{c}$.

A single inequality constraint $\mathbf{w}^\top \mathbf{x} \leq b$ forms a *half-space*, or a region on one side of a hyperplane. The hyperplane is perpendicular to $\mathbf{w}$ and is defined by $\mathbf{w}^\top \mathbf{x} = b$ as shown in figure 12.2. The region $\mathbf{w}^\top \mathbf{x} > b$ is on the $+\mathbf{w}$ side of the hyperplane, whereas $\mathbf{w}^\top \mathbf{x} < b$ is on the $-\mathbf{w}$ side of the hyperplane.

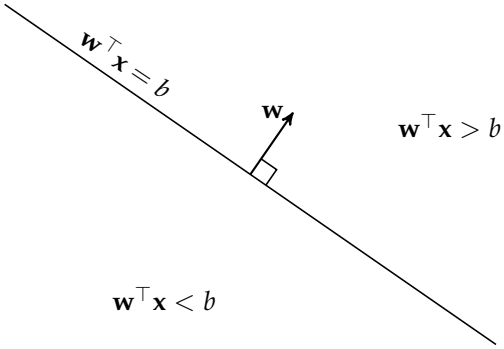


Figure 12.2. A linear constraint.

Half-spaces are convex sets (see appendix C.3), and the intersection of convex sets is convex, as shown in figure 12.3. Thus, the feasible set of a linear program will always form a convex set. Convexity of the feasible set, along with convexity of the objective function, implies that if we find a local feasible minimum, it is also a global feasible minimum.

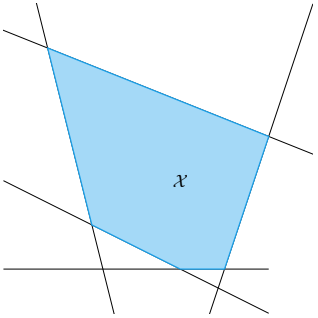


Figure 12.3. The intersection of linear constraints is a convex set.

The feasible set is a convex region enclosed by flat faces. Depending on the region's configuration, the solution can lie at a vertex, on an edge, or on an entire face. If the problem is not properly constrained, the solution can be unbounded, and, if the system is *over-constrained*, there is no feasible solution. Several such cases are shown in figure 12.4.

12.1.3 *Equality Form*

Linear programs are often solved in *equality form*:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{12.7}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

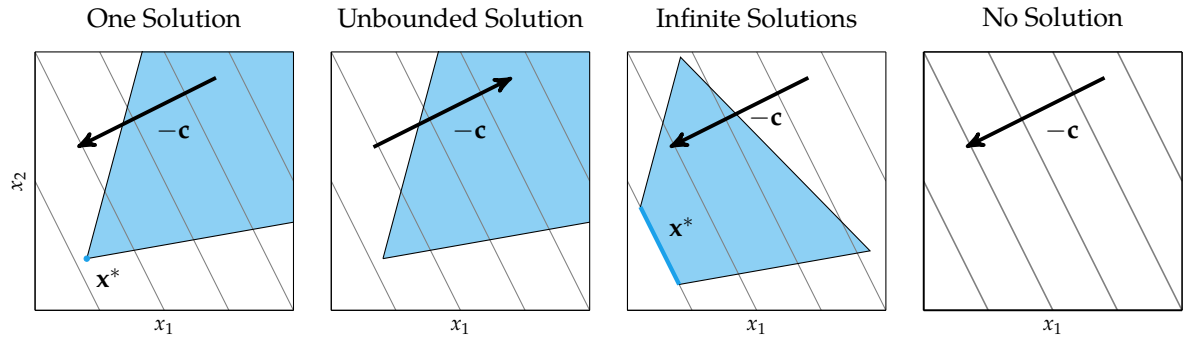


Figure 12.4. Several different linear problem forms with different solutions.

where $\mathbf{x}$ and $\mathbf{c}$ each have n components, $\mathbf{A}$ is an $m \times n$ matrix, and $\mathbf{b}$ has m components. In other words, we have n nonnegative design variables and a system of m equations defining equality constraints.

The equality form has constraints in two parts. The first, $\mathbf{Ax} = \mathbf{b}$, forces the solution to lie in an affine subspace.[5] Such a constraint is convenient because search techniques can constrain themselves to the constrained affine subspace to remain feasible. The second part of the constraints requires $\mathbf{x} \geq \mathbf{0}$, which forces the solution to lie in the nonnegative quadrant. The feasible set is thus the nonnegative portion of an affine subspace. Example 12.3 provides a visualization of a simple linear program.

[5] Informally, an *affine subspace* is a vector space that has been translated such that its origin in a higher-dimensional space is not necessarily $\mathbf{0}$.

Any linear program in standard form can be transformed to equality form. The constraints are converted as follows:

$$\mathbf{Ax} \leq \mathbf{b} \quad \rightarrow \quad \mathbf{Ax} + \mathbf{s} = \mathbf{b}, \quad \mathbf{s} \geq \mathbf{0} \tag{12.8}$$

by introducing slack variables $\mathbf{s}$. These variables take up the extra slack to enforce equality.

Starting with a linear program:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{12.9}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Consider the standard-form linear program:

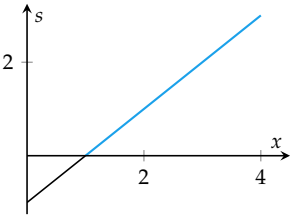
$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x \\ \text{subject to} \quad & x \geq 1 \end{aligned}$$

When we convert this to equality form, we get

$$\begin{aligned} \underset{x,s}{\text{minimize}} \quad & x \\ \text{subject to} \quad & x - s = 1 \\ & x, s \geq 0 \end{aligned}$$

The equality constraint requires that feasible points fall on the line $x - s = 1$. That line is a one-dimensional affine subspace of the two-dimensional Euclidean space.

Example 12.3. Feasible sets for the equality form are hyperplanes.



We introduce the slack variables:

$$\begin{aligned} \underset{\mathbf{x},\mathbf{s}}{\text{minimize}} \quad & \begin{bmatrix} \mathbf{c}^\top & \mathbf{0}^\top \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} \mathbf{A} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \end{bmatrix} = \mathbf{b} \\ & \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \end{bmatrix} \geq \mathbf{0} \end{aligned} \tag{12.10}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 12.4 demonstrates converting from standard to equality form.

Consider the linear program

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & 5x_1 + 4x_2 \\ \text{subject to} \quad & 2x_1 + 3x_2 \leq 5 \\ & 4x_1 + x_2 \leq 11 \end{aligned}$$

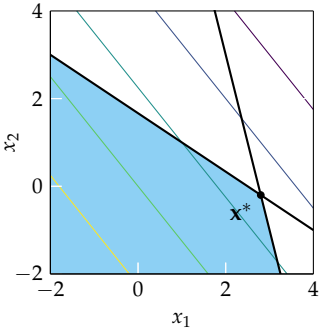
To convert to equality form, we first convert to minimization and introduce two slack variables:

$$\begin{aligned} \underset{\mathbf{x},\mathbf{s}}{\text{minimize}} \quad & -5x_1 - 4x_2 \\ \text{subject to} \quad & 2x_1 + 3x_2 + s_1 = 5 \\ & 4x_1 + x_2 + s_2 = 11 \\ & s_1, s_2 \geq 0 \end{aligned}$$

We then split $\mathbf{x}$:

$$\begin{aligned} \underset{\mathbf{x}^+,\mathbf{x}^-,\mathbf{s}}{\text{minimize}} \quad & -5(x_1^+ - x_1^-) - 4(x_2^+ - x_2^-) \\ \text{subject to} \quad & 2(x_1^+ - x_1^-) + 3(x_2^+ - x_2^-) + s_1 = 5 \\ & 4(x_1^+ - x_1^-) + (x_2^+ - x_2^-) + s_2 = 11 \\ & x_1^+, x_1^-, x_2^+, x_2^-, s_1, s_2 \geq 0 \end{aligned}$$

Example 12.4. Converting a linear program to equality form. Below we show the original linear program. The linear program in equality form has 6 dimensions, making it more difficult to visualize.



12.2 Simplex Algorithm

The *simplex algorithm* solves linear programs by moving from vertex to vertex of the feasible set.[6] The method is guaranteed to arrive at an optimal solution so long as the linear program is feasible and bounded.

The simplex algorithm operates on equality-form linear programs ($\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$). We assume that the rows of $\mathbf{A}$ are linearly independent.[7] We also assume that the problem has no more equality constraints than it has design variables ($m \leq n$), which ensures that the problem is not over constrained. A preprocessing phase guarantees that $\mathbf{A}$ satisfies these conditions.

[6] The simplex algorithm was originally developed in the 1940s by George Dantzig. A history of the development can be found here: G. B. Dantzig, "Origins of the Simplex Method," in *A History of Scientific Computing*, S. G. Nash, ed., ACM, 1990, pp. 141–151.

[7] A matrix whose rows are linearly independent is said to have full row rank. Linear independence is achieved by removing redundant equality constraints.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

12.2.1 *Vertices*

Linear programs in equality form have feasible sets in the form of convex *polytopes*, which are geometric objects with flat faces. These polytopes are formed by the intersection of the equality constraints with the nonnegative quadrant. Associated with a polytope are *vertices*, which are points in the feasible set that do not lie between any other points in the feasible set.

The feasible set consists of several different types of design points. Points on the interior are never optimal because they can be improved by moving along $-\mathbf{c}$. Points on faces can be optimal only if the face is perpendicular to $\mathbf{c}$. Points on faces not perpendicular to $\mathbf{c}$ can be improved by sliding along the face in the direction of the projection of $-\mathbf{c}$ onto the face. Similarly, points on edges can be optimal only if the edge is perpendicular to $\mathbf{c}$, and can otherwise be improved by sliding along the projection of $-\mathbf{c}$ onto the edge. Finally, vertices can also be optimal.

The simplex algorithm produces an optimal vertex. If a linear program has a bounded solution, then it also contains at least one vertex. Furthermore, at least one solution must lie at a vertex. In the case where an entire edge or face is optimal, a vertex solution is just as good as any other.

Every vertex for a linear program in equality form can be uniquely defined by $n - m$ components of $\mathbf{x}$ that equal zero.[8] These components are actively constrained by the nonnegative quadrant requirement.

The equality constraint $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution when $\mathbf{A}$ is square. We have assumed that $m \leq n$, so choosing m design variables and setting the remaining variables to zero effectively removes $n - m$ columns of $\mathbf{A}$, yielding an $m \times m$ constraint matrix (see example 12.5).

[8] This is analogous to identifying a vertex in standard form with $n - m$ active constraints, including original equality constraints and a set of active inequality constraints necessary to uniquely identify a vertex. Zeroing any slack variables associated with standard form inequality constraints causes those original inequality constraints to be active.

Example 12.5. Setting $n - m$ components of $\mathbf{x}$ to zero can uniquely define a point.

For a problem with 5 design variables and 3 constraints, setting 2 variables to zero uniquely defines a point.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \end{bmatrix} \begin{bmatrix} x_1 \\ 0 \\ x_3 \\ x_4 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The indices into the components $1:n$ of any vertex can be partitioned into two sets, $\mathcal{B}$ and $\mathcal{V}$, such that:[9]

- The design values associated with indices in $\mathcal{V}$ are zero:

$$i \in \mathcal{V} \implies x_i = 0 \tag{12.11}$$

- The design values associated with indices in $\mathcal{B}$ may or may not be zero:

$$i \in \mathcal{B} \implies x_i \geq 0 \tag{12.12}$$

- $\mathcal{B}$ has exactly m elements and $\mathcal{V}$ has exactly $n - m$ elements.

[9] Sometimes $\mathcal{B}$ is referred to as the set of *basic* indices, in the sense that the corresponding variables are part of the current solution basis. The set $\mathcal{V}$ contains the *non-basic* indices, whose corresponding variables are set to zero. A mnemonic that we can use here is that the indices in $\mathcal{B}$ are "busy" in the sense that we will be using them to solve the linear system, and that the indices in $\mathcal{V}$ are vacant in the sense that they are set to zero.

We use $\mathbf{x}_\mathcal{B}$ to refer to the vector consisting of the components of $\mathbf{x}$ that are in $\mathcal{B}$ and $\mathbf{x}_\mathcal{V}$ to refer to the vector consisting of the components of $\mathbf{x}$ that are in $\mathcal{V}$. Note that $\mathbf{x}_\mathcal{V} = \mathbf{0}$.

The vertex associated with a partition $(\mathcal{B}, \mathcal{V})$ can be obtained using the $m \times m$ matrix $\mathbf{A}_\mathcal{B}$ formed by the m columns of $\mathbf{A}$ selected by $\mathcal{B}$:[10]

$$\mathbf{A}\mathbf{x} = \mathbf{A}_\mathcal{B}\mathbf{x}_\mathcal{B} = \mathbf{b} \quad \rightarrow \quad \mathbf{x}_\mathcal{B} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b} \tag{12.13}$$

[10] If $\mathcal{B}$ and $\mathcal{V}$ identify a vertex, then the columns of $\mathbf{A}_\mathcal{B}$ must be linearly independent because $\mathbf{A}\mathbf{x} = \mathbf{b}$ must have exactly one solution. Hence, $\mathbf{A}_\mathcal{B}\mathbf{x}_\mathcal{B} = \mathbf{b}$ must have exactly one solution. This linear independence guarantees that $\mathbf{A}_\mathcal{B}$ is invertible.

Knowing $\mathbf{x}_\mathcal{B}$ is sufficient to construct $\mathbf{x}$; the remaining design variables are zero. Algorithm 12.1 implements this procedure, and example 12.6 demonstrates verifying that a given design point is a vertex.

```
struct LinearProgram
    A # LP in equality form:
    b #  minimize x⋅c
    c #  subject to Ax = b
      #              x ≥ 0
end
function get_vertex(B, LP)
    A, b, c = LP.A, LP.b, LP.c
    b_inds = sort(B)
    AB = A[:,b_inds]
    xB = AB\b
    x = zero(c)
    x[b_inds] = xB
    return x
end
```

Algorithm 12.1. A method for extracting the vertex associated with a partition `B` and a linear program `LP` in equality form. We introduce the special type `LinearProgram` for linear programs in equality form.

While every vertex has an associated partition $(\mathcal{B}, \mathcal{V})$, not every partition corresponds to a vertex. A partition corresponds to a vertex only if $\mathbf{A}_\mathcal{B}$ is nonsingular

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and the design obtained by applying equation (12.13) is feasible.[11] Identifying partitions that correspond to vertices is nontrivial, and we show in section 12.2.4 that finding such a partition involves solving a linear program. The simplex algorithm operates in two phases—an *initialization phase* that identifies a vertex partition and an *optimization phase* that transitions between vertex partitions toward a partition corresponding to an optimal vertex. We will discuss both of these phases later in this section.

[11] For example, $\mathcal{B} = \{1, 2\}$ for the constraints

$$\begin{bmatrix} 1 & 2 & 0 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

corresponds to

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which does not produce an invertible $\mathbf{A}_\mathcal{B}$ and does not have a unique solution.

Example 12.6. Verifying that a design point is a vertex for constraints in equality form.

Consider the constraints:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & 2 & 3 \\ 2 & 1 & 2 & -1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}, \quad \mathbf{x} \geq \mathbf{0}$$

Consider the design point $\mathbf{x} = [1, 1, 0, 0]$. We can verify that $\mathbf{x}$ is feasible and that it has no more than three nonzero components. We can choose either $\mathcal{B} = \{1, 2, 3\}$ or $\mathcal{B} = \{1, 2, 4\}$. Both

$$\mathbf{A}_{\{1,2,3\}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

and

$$\mathbf{A}_{\{1,2,4\}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 3 \\ 2 & 1 & -1 \end{bmatrix}$$

are invertible. Thus, $\mathbf{x}$ is a vertex of the feasible set polytope.

12.2.2 First-Order Necessary Conditions

The first-order necessary conditions for optimality are used to determine when a vertex is optimal and to inform how to transition to a more favorable vertex. We construct a Lagrangian for the equality form of the linear program:[12]

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = \mathbf{c}^\top \mathbf{x} - \boldsymbol{\mu}^\top \mathbf{x} - \boldsymbol{\lambda}^\top (\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{12.14}$$

with the following necessary conditions:

[12] Note that in $\mathbf{x} \geq 0$ the polarity of the inequality must be inverted by multiplying both sides by -1, yielding the negative sign in front of $\boldsymbol{\mu}$. The Lagrangian can be defined with either positive or negative $\boldsymbol{\lambda}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

1. **feasibility:** $\mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$
2. **dual feasibility:** $\boldsymbol{\mu} \geq \mathbf{0}$
3. **complementary slackness:** $\boldsymbol{\mu} \odot \mathbf{x} = \mathbf{0}$
4. **stationarity:** $\mathbf{A}^\top \boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c}$

The necessary conditions are also sufficient conditions for optimality for linear programs. Thus, if $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ can be computed for a given vertex and all four necessary conditions are satisfied, then the vertex is optimal.

We can decompose the stationarity condition into $\mathcal{B}$ and $\mathcal{V}$ components:

$$\mathbf{A}^\top \boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c} \quad \rightarrow \quad \begin{cases} \mathbf{A}_\mathcal{B}^\top \boldsymbol{\lambda} + \boldsymbol{\mu}_\mathcal{B} = \mathbf{c}_\mathcal{B} \\ \mathbf{A}_\mathcal{V}^\top \boldsymbol{\lambda} + \boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} \end{cases} \tag{12.15}$$

We can choose $\boldsymbol{\mu}_\mathcal{B} = \mathbf{0}$ to satisfy complementary slackness. The value of $\boldsymbol{\lambda}$ can be computed from $\mathcal{B}$:[13]

$$\mathbf{A}_\mathcal{B}^\top \boldsymbol{\lambda} + \underbrace{\boldsymbol{\mu}_\mathcal{B}}_{=\mathbf{0}} = \mathbf{c}_\mathcal{B} \tag{12.16}$$

$$\boldsymbol{\lambda} = \mathbf{A}_\mathcal{B}^{-\top} \mathbf{c}_\mathcal{B} \tag{12.17}$$

We can use this to obtain

$$\mathbf{A}_\mathcal{V}^\top \boldsymbol{\lambda} + \boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} \tag{12.18}$$

$$\boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} - \mathbf{A}_\mathcal{V}^\top \boldsymbol{\lambda} \tag{12.19}$$

$$\boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} - \left(\mathbf{A}_\mathcal{B}^{-1} \mathbf{A}_\mathcal{V}\right)^\top \mathbf{c}_\mathcal{B} \tag{12.20}$$

Knowing $\boldsymbol{\mu}_\mathcal{V}$ allows us to assess the optimality of the vertices. If $\boldsymbol{\mu}_\mathcal{V}$ contains negative components, then dual feasibility is not satisfied and the vertex is suboptimal.

12.2.3 *Optimization Phase*

The simplex algorithm maintains a partition $(\mathcal{B}, \mathcal{V})$, which corresponds to a vertex of the feasible set polytope. The partition can be updated by swapping indices between $\mathcal{B}$ and $\mathcal{V}$.[14] Such a swap equates to moving from one vertex along an edge of the feasible set polytope to another vertex. If the initial partition corresponds to a vertex and the problem is bounded, the simplex algorithm is guaranteed to converge to an optimum.

[13] We use $\mathbf{A}^{-\top}$ to refer to the transpose of the inverse of $\mathbf{A}$:

$$\mathbf{A}^{-\top} = \left(\mathbf{A}^{-1}\right)^\top = \left(\mathbf{A}^\top\right)^{-1}$$

[14] The vertex given by $\mathcal{B} = \{1, 2, 3\}$ and $\mathcal{V} = \{4\}$ in example 12.6 can be moved to the vertex given by $\mathcal{B} = \{1, 2, 4\}$ and $\mathcal{V} = \{3\}$ by swapping indices 3 and 4 between $\mathcal{B}$ and $\mathcal{V}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A transition $\mathbf{x} \to \mathbf{x}'$ between vertices must satisfy $\mathbf{A}\mathbf{x}' = \mathbf{b}$. Starting with a partition defined by $\mathcal{B}$, we choose an *entering index* $q \in \mathcal{V}$ that is to enter $\mathcal{B}$ using one of the heuristics described near the end of this section. The new vertex $\mathbf{x}'$ must satisfy:

$$\mathbf{A}\mathbf{x}' = \mathbf{A}_{\mathcal{B}}\mathbf{x}'_{\mathcal{B}} + \mathbf{A}_{\{q\}}x'_q = \mathbf{A}_{\mathcal{B}}\mathbf{x}_{\mathcal{B}} = \mathbf{A}\mathbf{x} = \mathbf{b} \tag{12.21}$$

Originally, the value at the entering index $q \in \mathcal{V}$ is zero. To obtain x'_q, we increase x_q until the component of $\mathbf{x}'$ corresponding to the *leaving index* $p \in \mathcal{B}$ becomes zero, after which p can be moved to $\mathcal{V}$. We do not increase x_q any further because it could result in other components in $\mathcal{B}$ becoming negative. This action is referred to as *pivoting*.

We can solve for the new design point

$$\mathbf{x}'_{\mathcal{B}} = \mathbf{x}_{\mathcal{B}} - \mathbf{A}_{\mathcal{B}}^{-1}\mathbf{A}_{\{q\}}x'_q \tag{12.22}$$

A particular leaving index $p \in \mathcal{B}$ becomes active when:[15]

$$\left(\mathbf{x}'_{\mathcal{B}}\right)_p = 0 = \left(\mathbf{x}_{\mathcal{B}}\right)_p - \left(\mathbf{A}_{\mathcal{B}}^{-1}\mathbf{A}_{\{q\}}\right)_p x'_q \tag{12.23}$$

and is thus obtained by increasing $x_q = 0$ to x'_q with:

$$x'_q = \frac{\left(\mathbf{x}_{\mathcal{B}}\right)_p}{\left(\mathbf{A}_{\mathcal{B}}^{-1}\mathbf{A}_{\{q\}}\right)_p} \tag{12.24}$$

The leaving index p is selected from $\mathcal{B}$ so that it increases the value of the variable with entering index q the least. Since this new value of the variable associated with index q is given by the ratio shown in equation (12.24), this selection criterion for the leaving index p is referred to as the *minimum ratio test*. The minimum ratio test computes equation (12.24) for each potential leaving index and selects the one with the minimum x'_q. We then swap p and q between $\mathcal{B}$ and $\mathcal{V}$. This edge transition is implemented in algorithm 12.2.

The effect that a transition has on the objective function can be computed using x'_q. The objective function value at the new vertex is

$$\mathbf{c}^\top\mathbf{x}' = \mathbf{c}_{\mathcal{B}}^\top\mathbf{x}'_{\mathcal{B}} + c_q x'_q \tag{12.25}$$

We apply equation (12.22) and expand:

$$\mathbf{c}^\top\mathbf{x}' = \mathbf{c}_{\mathcal{B}}^\top\left(\mathbf{x}_{\mathcal{B}} - \mathbf{A}_{\mathcal{B}}^{-1}\mathbf{A}_{\{q\}}x'_q\right) + c_q x'_q \tag{12.26}$$

$$= \mathbf{c}_{\mathcal{B}}^\top\mathbf{x}_{\mathcal{B}} - \mathbf{c}_{\mathcal{B}}^\top\mathbf{A}_{\mathcal{B}}^{-1}\mathbf{A}_{\{q\}}x'_q + c_q x'_q \tag{12.27}$$

[15] In the next two equations, we use $(\cdot)_p$ to refer to the element associated with $p \in \mathcal{B}$, which is not necessarily the pth element of the input vector.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function edge_transition(LP, B, q)
    A, b = LP.A, LP.b
    n = size(A, 2)
    b_inds = sort(B)
    n_inds = sort(setdiff(1:n, B))
    AB = A[:,b_inds]
    d, xB = AB\A[:,n_inds[q]], AB\b

    best = (p=0, xq′=Inf)
    for p in eachindex(d)
        if d[p] > 0
            v = xB[p] / d[p]
            if v < best.xq′
                best = (p=p, xq′=v)
            end
        end
    end
    return best
end
```

Algorithm 12.2. A method for computing the index p and the new coordinate value x'_q obtained by increasing index `q` of the vertex defined by the partition `B` in the equality-form linear program `LP`.

By applying equation (12.18) to the entering index, we obtain

$$\mathbf{A}_{\{q\}}^\top \boldsymbol{\lambda} + \mu_q = c_q \tag{12.28}$$

We can replace $\boldsymbol{\lambda}$ using equation (12.17) and rearrange, yielding:

$$\mathbf{A}_{\{q\}}^\top \mathbf{A}_{\mathcal{B}}^{-\top} \mathbf{c}_{\mathcal{B}} = \mathbf{c}_{\mathcal{B}}^\top \mathbf{A}_{\mathcal{B}}^{-1} \mathbf{A}_{\{q\}} = c_q - \mu_q \tag{12.29}$$

Substituting equation (12.29) into equation (12.27) yields:

$$\begin{aligned} \mathbf{c}^\top \mathbf{x}' &= \mathbf{c}_{\mathcal{B}}^\top \mathbf{x}_{\mathcal{B}} - (c_q - \mu_q) x'_q + c_q x'_q && (12.30)\\ &= \mathbf{c}^\top \mathbf{x} + \mu_q x'_q && (12.31) \end{aligned}$$

Choosing an entering index q thus decreases the objective function value by

$$\mathbf{c}^\top \mathbf{x}' - \mathbf{c}^\top \mathbf{x} = \mu_q x'_q \tag{12.32}$$

The objective function decreases only if μ_q is negative. In order to progress toward optimality, we must choose an index q in $\mathcal{V}$ such that μ_q is negative. If all components of $\boldsymbol{\mu}_{\mathcal{V}}$ are nonnegative, we have found a global optimum.

Since there can be multiple negative entries in $\boldsymbol{\mu}_{\mathcal{V}}$, different heuristics can be used to select an entering index:[16]

- *Greedy heuristic*, which chooses a q that maximally reduces $\mathbf{c}^\top \mathbf{x}$.

[16] Modern implementations use more sophisticated rules. For example, see J. J. Forrest and D. Goldfarb, "Steepest-Edge Simplex Algorithms for Linear Programming," *Mathematical Programming*, vol. 57, no. 1, pp. 341–374, 1992.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- *Dantzig's rule*, which chooses the q with the most negative entry in $\boldsymbol{\mu}$. This rule is easy to calculate, but it does not guarantee the maximum reduction in $\mathbf{c}^\top \mathbf{x}$. It is also sensitive to scaling of the constraints.[17]
- *Bland's rule*, which chooses the first q with a negative entry in $\boldsymbol{\mu}$. When used on its own, Bland's rule tends to result in poor performance in practice. However, this rule can help us prevent *cycles*, which occur when we return to a vertex we have visited before without decreasing the objective function. This rule is usually used only after no improvements have been made for several iterations of a different rule to break out of a cycle and ensure convergence.

[17] For constraint $\mathbf{A}^\top \mathbf{x} = \mathbf{b} \rightarrow \alpha \mathbf{A}^\top \mathbf{x} = \alpha \mathbf{b}$, $\alpha > 0$, we do not change the solution but the Lagrange multipliers are scaled, $\boldsymbol{\lambda} \rightarrow \alpha^{-1} \boldsymbol{\lambda}$.

One iteration of the simplex method's optimization phase moves a vertex partition to a neighboring vertex based on a heuristic for the entering index. Algorithm 12.3 implements such an iteration with the greedy heuristic. Example 12.7 demonstrates using the simplex algorithm starting from a known vertex partition to solve a linear program.

12.2.4 Initialization Phase

The *optimization phase* of the simplex algorithm is implemented in algorithm 12.4. Unfortunately, algorithm 12.4 requires an initial partition that corresponds to a vertex. If we do not have such a partition, we must solve an *auxiliary linear program* to obtain this partition as part of an *initialization phase*.

The auxiliary linear program to be solved in the initialization phase includes extra variables $\mathbf{z} \in \mathbb{R}^m$, which we seek to zero out:

$$\begin{aligned} \underset{\mathbf{x},\mathbf{z}}{\text{minimize}} \quad & \begin{bmatrix} \mathbf{0}^\top & \mathbf{1}^\top \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} \mathbf{A} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} = \mathbf{b} \\ & \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix} \geq \mathbf{0} \end{aligned} \tag{12.33}$$

where $\mathbf{Z}$ is a diagonal matrix whose diagonal entries are

$$Z_{ii} = \begin{cases} +1 & \text{if } b_i \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{12.34}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 12.7. Solving a linear program with the simplex algorithm.

Consider the equality-form linear program with

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ -4 & 2 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 9 \\ 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 3 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

and the initial vertex defined by $\mathcal{B} = \{3, 4\}$. After verifying that $\mathcal{B}$ defines a feasible vertex, we can begin one iteration of the simplex algorithm.

We extract $\mathbf{x}_\mathcal{B}$:

$$\mathbf{x}_\mathcal{B} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 9 \\ 2 \end{bmatrix} = \begin{bmatrix} 9 \\ 2 \end{bmatrix}$$

We then compute $\boldsymbol{\lambda}$:

$$\boldsymbol{\lambda} = \mathbf{A}_\mathcal{B}^{-\top}\mathbf{c}_\mathcal{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-\top} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{0}$$

and $\boldsymbol{\mu}_\mathcal{V}$:

$$\boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} - \left(\mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_\mathcal{V}\right)^\top \mathbf{c}_\mathcal{B} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} - \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 \\ -4 & 2 \end{bmatrix}\right)^\top \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

Because $\boldsymbol{\mu}_\mathcal{V}$ contains negative elements, our current $\mathcal{B}$ is suboptimal. We will pivot on the index of the only negative element, $q = 2$. An edge transition is run from $\mathbf{x}_\mathcal{B}$ in the direction $-\mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_{\{q\}} = -[1, 2]$.

Using equation (12.22), we increase x'_q until a new constraint becomes active. In this case, $x'_q = 1$ causes x_4 to become zero. We update $\mathcal{B}$ to $\{2, 3\}$.

In the second iteration, we find:

$$\mathbf{x}_\mathcal{B} = \begin{bmatrix} 1 \\ 8 \end{bmatrix}, \quad \boldsymbol{\lambda} = \begin{bmatrix} 0 \\ -1/2 \end{bmatrix}, \quad \boldsymbol{\mu}_\mathcal{V} = \begin{bmatrix} 1 \\ 1/2 \end{bmatrix}.$$

The vertex is optimal because $\boldsymbol{\mu}_\mathcal{V}$ has no negative entries. Our algorithm thus terminates with $\mathcal{B} = \{2, 3\}$, for which the design point is $\mathbf{x}^* = [0, 1, 8, 0]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function step_lp!(B, LP)
    A, b, c = LP.A, LP.b, LP.c
    n = size(A, 2)
    b_inds = sort(B)
    n_inds = sort(setdiff(1:n, B))
    AB, AV = A[:,b_inds], A[:,n_inds]
    xB = AB\b
    cB = c[b_inds]
    λ = AB' \ cB
    cV = c[n_inds]
    μV = cV - AV'*λ

    best = (q=0, p=0, xq′=Inf, Δ=Inf)
    for q in eachindex(μV)
        if μV[q] < 0
            p, xq′ = edge_transition(LP, B, q)
            if μV[q]*xq′ < best.Δ
                best = (q=q, p=p, xq′=xq′, Δ=μV[q]*xq′)
            end
        end
    end
    if best.q == 0
        return (B, true) # optimal point found
    end

    if isinf(best.xq′) || best.xq′ < -eps()
        error("unbounded")
    end

    p, q = best.p, best.q
    j = findfirst(isequal(b_inds[p]), B)
    B[j] = n_inds[q] # swap indices
    return (B, false) # new vertex but not optimal
end
```

Algorithm 12.3. A single iteration of the simplex algorithm in which the set $\mathcal{B}$ is moved from one vertex to a neighbor while maximally decreasing the objective function. Here, `step_lp!` takes a partition `B` and a linear program `LP`.

```
function minimize_lp!(B, LP)
    done = false
    while !done
        B, done = step_lp!(B, LP)
    end
    return B
end
```

Algorithm 12.4. Minimizing a linear program given an initial vertex partition defined by `B` and a linear program `LP`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The values for $\mathbf{z}$ represent the amount by which $\mathbf{A}\mathbf{x} = \mathbf{b}$ is violated. By zeroing out $\mathbf{z}$, we find a feasible point. If, in solving the auxiliary problem, we do not find a vertex with a zeroed-out $\mathbf{z}$, then we can conclude that the problem is infeasible.

This auxiliary linear program is solved using algorithm 12.4 with an initial partition that selects only the $\mathbf{z}$ components. The corresponding vertex has $\mathbf{x} = \mathbf{0}$, and each element in $\mathbf{z}$ is set to the absolute value of the corresponding value in $\mathbf{b}$.[18] Example 12.8 demonstrates using an auxiliary linear program to obtain a feasible vertex.

[18] In other words, $z_j = |b_j|$.

Example 12.8. Using an auxiliary linear program to obtain a feasible vertex.

Consider the equality-form linear program:

$$\begin{aligned} \underset{x_1,x_2,x_3}{\text{minimize}} \quad & c_1x_1 + c_2x_2 + c_3x_3 \\ \text{subject to} \quad & 2x_1 - 1x_2 + 2x_3 = 1 \\ & 5x_1 + 1x_2 - 3x_3 = -2 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

We can identify a feasible vertex by solving:

$$\begin{aligned} \underset{x_1,x_2,x_3,z_1,z_2}{\text{minimize}} \quad & z_1 + z_2 \\ \text{subject to} \quad & 2x_1 - 1x_2 + 2x_3 + z_1 = 1 \\ & 5x_1 + 1x_2 - 3x_3 - z_2 = -2 \\ & x_1, x_2, x_3, z_1, z_2 \geq 0 \end{aligned}$$

with an initial vertex defined by $\mathcal{B} = \{4, 5\}$.

The initial vertex has:

$$\mathbf{x}_{\mathcal{B}}^{(1)} = \mathbf{A}_{\mathcal{B}}^{-1}\mathbf{b} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

and is thus $\mathbf{x}^{(1)} = [0, 0, 0, 1, 2]$. Solving the auxiliary problem yields $\mathbf{x}^* \approx [0, 1, 1, 0, 0]$ and the vertex partition $\mathcal{B} = \{2, 3\}$. Thus $[0, 1, 1]$ is a feasible vertex in the original problem.

The partition $\mathcal{B}$ can only be used to solve the original linear program if its indices all correspond to entries in $\mathbf{x}$. It is possible that some indices in the compo-

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

nents of $\mathbf{z}$ are included in the partition. This can happen when the initialization phase finds a feasible vertex where some components of $\mathbf{x}$ are zero. We can replace any entries in $\mathcal{B}$ that correspond to $\mathbf{z}$ with unused indices that correspond to $\mathbf{x}$.

Algorithm 12.5 implements the complete simplex algorithm.

```
function find_partition(LP)
    A, b, c = LP.A, LP.b, LP.c
    m, n = size(A)
    Z = Diagonal([j ≥ 0 ? 1 : -1 for j in b])

    A′ = [A Z]
    b′ = b
    c′ = [zeros(n); ones(m)]
    LP_init = LinearProgram(A′, b′, c′)
    B = collect(1:m).+n
    minimize_lp!(B, LP_init)

    # Scan through the vertex partition and replace
    # entries corresponding to z-values with x-values
    available_xs = setdiff(1:n, B)
    zs_to_replace = collect(findall(x -> x > n, B))
    B[zs_to_replace] = available_xs[1:length(zs_to_replace)]
    sort!(B)

    return B
end

function minimize_lp(LP)
    B = find_partition(LP)
    minimize_lp!(B, LP)
    return (x=get_vertex(B, LP), B)
end
```

Algorithm 12.5. The simplex algorithm for solving linear programs in equality form when an initial partition is not known. We first construct and solve an auxiliary linear program in order to find a vertex partition, and then that vertex partition is used to solve the original linear program. This method returns both the optimal design and its partition.

12.3 *Dual Certificates*

Linear programs are linear and convex, and can be shown to have zero duality gap. The zero duality gap implies that the optimal value of the dual problem d^* is equal to the optimal value of the primal problem p^*. The inclusion of optimal dual variables for a candidate primal solution $\mathbf{x}^\star$ can be used to verify its optimality. Verifying optimality using *dual certificates* (algorithm 12.6) is useful in many cases, such as debugging our linear program code.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The primal linear program can be converted to its dual form:

$$\max_{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x},\boldsymbol{\mu},\boldsymbol{\lambda}) = \max_{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}} \min_{\mathbf{x}} \mathbf{c}^\top\mathbf{x} - \boldsymbol{\mu}^\top\mathbf{x} - \boldsymbol{\lambda}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) \tag{12.35}$$

$$= \max_{\boldsymbol{\mu}\geq\mathbf{0},\boldsymbol{\lambda}} \min_{\mathbf{x}} (\mathbf{c}-\boldsymbol{\mu}-\mathbf{A}^\top\boldsymbol{\lambda})^\top\mathbf{x} + \boldsymbol{\lambda}^\top\mathbf{b} \tag{12.36}$$

From the first-order necessary conditions, we know $\mathbf{c}-\boldsymbol{\mu}-\mathbf{A}^\top\boldsymbol{\lambda} = \mathbf{0}$, which allows us to drop the first term in the objective above. In addition, we know $\boldsymbol{\mu} = \mathbf{c}-\mathbf{A}^\top\boldsymbol{\lambda} \geq \mathbf{0}$, which implies $\mathbf{A}^\top\boldsymbol{\lambda} \leq \mathbf{c}$. In summary, we have:

Primal Form (equality)		Dual Form	
$\underset{\mathbf{x}}{\text{minimize}}$	$\mathbf{c}^\top\mathbf{x}$	$\underset{\boldsymbol{\lambda}}{\text{maximize}}$	$\mathbf{b}^\top\boldsymbol{\lambda}$
subject to	$\mathbf{A}\mathbf{x} = \mathbf{b}$	subject to	$\mathbf{A}^\top\boldsymbol{\lambda} \leq \mathbf{c}$
	$\mathbf{x} \geq \mathbf{0}$		

If the primal problem has n variables and m equality constraints, then the dual problem has m variables and n constraints.[19] Furthermore, the dual of the dual is the primal problem.

Optimality can be assessed by verifying three properties. If someone claims $(\mathbf{x}^*, \boldsymbol{\lambda}^*)$ is optimal, we can quickly verify the claim by checking whether all three of the following conditions are satisfied:

1. $\mathbf{x}^*$ is feasible in the primal problem.
2. $\boldsymbol{\lambda}^*$ is feasible in the dual problem.
3. $p^* = \mathbf{c}^\top\mathbf{x}^* = \mathbf{b}^\top\boldsymbol{\lambda}^* = d^*$.

Dual certificates are used in example 12.9 to verify the solution to a linear program.

[19] An alternative to the simplex algorithm, the *self-dual simplex algorithm*, tends to be faster in practice. It does not require that the matrix $\mathbf{A}_\mathcal{B}$ satisfy $\mathbf{x}_\mathcal{B} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b} \geq \mathbf{0}$. The self-dual simplex algorithm is a modification of the simplex algorithm for the dual of the linear programming problem in standard form.

```
function dual_certificate(LP, x, λ, ϵ=1e-6)
    A, b, c = LP.A, LP.b, LP.c
    primal_feasible = all(x .≥ 0) && A*x ≈ b
    dual_feasible = all(A'*λ .≤ c)
    return primal_feasible && dual_feasible &&
           isapprox(c⋅x, b⋅λ, atol=ϵ)
end
```

Algorithm 12.6. A method for checking whether a candidate solution given by design point `x` and dual point `λ` for the linear program `LP` in equality form is optimal. The parameter `ϵ` controls the tolerance for the equality constraint.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 12.9. Verifying a solution using dual certificates.

Consider the equality-form linear program with

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & -1 \\ -1 & 2 & 0 \\ 1 & 2 & 3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ 5 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$$

We would like to determine whether $\mathbf{x}^* = [2, 0, 1]$ and $\boldsymbol{\lambda}^* = [1, 0, 0]$ are an optimal solution pair. We first verify that $\mathbf{x}^*$ is feasible:

$$\mathbf{A}\mathbf{x}^* = [1, -2, 5] = \mathbf{b}, \qquad \mathbf{x}^* \geq \mathbf{0}$$

We then verify that $\boldsymbol{\lambda}^*$ is dual-feasible:

$$\mathbf{A}^\top \boldsymbol{\lambda}^* = [1, 1, -1] \leq \mathbf{c}$$

Finally, we verify that p^* and d^* are the same:

$$p^* = \mathbf{c}^\top \mathbf{x}^* = 1 = \mathbf{b}^\top \boldsymbol{\lambda}^* = d^*$$

We conclude that $(\mathbf{x}^*, \boldsymbol{\lambda}^*)$ are optimal.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

12.4 Summary

- Linear programs are problems consisting of a linear objective function and linear constraints.
- The simplex algorithm can optimize linear programs globally in an efficient manner.
- A vertex is a feasible point that satisfies a set of linear constraints.
- The simplex algorithm iteratively moves from one vertex to an adjacent vertex, improving the objective function with each step.
- The initialization phase of the simplex algorithm solves an auxiliary linear program to find an initial vertex.
- Dual certificates allow us to verify that a candidate primal-dual solution pair is optimal.

12.5 Exercises

Exercise 12.1. Suppose you do not know any optimization algorithm for solving a linear program. You decide to evaluate all the vertices and determine, by inspection, which one minimizes the objective function. Give a loose upper bound on the number of possible minimizers you will examine. Furthermore, does this method properly handle all linear-constrained optimization problems?

Solution: We have chosen to minimize a linear program by evaluating every vertex in the convex polytope formed by the constraints. Every vertex is thus a potential minimizer. Vertices are defined by intersections of active constraints. As every inequality constraint can either be active or inactive, and assuming there are n inequality constraints, we do not need to examine more than 2^n combinations of constraints.

This method does not correctly report unbounded linear-constrained optimization problems as unbounded.

Exercise 12.2. If the program in example 12.1 is bounded below, argue that the simplex method must converge.

Solution: The simplex method is guaranteed either to improve with respect to the objective function with each step or to preserve the current value of the objective function. Any linear program will have a finite number of vertices. So long as a heuristic, such as Bland's rule, is employed such that cycling does not occur, the simplex method must converge on a solution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 12.3. Suppose we want to minimize $6x_1 + 5x_2$ subject to the constraint $3x_1 - 2x_2 \geq 5$. How would we translate this problem into a linear program in equality form with the same minimizer?

Solution: The linear program was given to us in general form. To transform it to equality form, we first invert the inequality constraint:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & 6x_1 + 5x_2 \\ \text{subject to} \quad & -3x_1 + 2x_2 \leq -5 \end{aligned}$$

We then split our design variables by sign to obtain standard form:

$$\begin{aligned} \underset{x_1^\pm,x_2^\pm}{\text{minimize}} \quad & 6x_1^+ + 5x_2^+ - 6x_1^- - 5x_2^- \\ \text{subject to} \quad & -3x_1^+ + 2x_2^+ + 3x_1^- - 2x_2^- \leq -5 \\ & x_1^\pm, x_2^\pm \geq 0 \end{aligned}$$

Finally, we add a slack variable $x_3 \geq 0$ to obtain equality form:

$$\begin{aligned} \underset{x_1^\pm,x_2^\pm,x_3}{\text{minimize}} \quad & 6x_1^+ + 5x_2^+ - 6x_1^- - 5x_2^- \\ \text{subject to} \quad & -3x_1^+ + 2x_2^+ + 3x_1^- - 2x_2^- + x_3 = -5 \\ & x_1^\pm, x_2^\pm, x_3 \geq 0 \end{aligned}$$

Exercise 12.4. Consider the equality-form linear program with

$$\mathbf{A} = \begin{bmatrix} -1 & -1 & 1 & 0 & 0 & 0 \\ 1 & -2 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ -1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -1 \\ 1 \\ 7 \\ 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 1 \\ 3 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and the initial vertex defined by $\mathcal{B} = \{1, 2, 3, 4\}$. Verify that $\mathcal{B}$ defines a feasible vertex, and then execute one iteration of the simplex algorithm.

Solution: We can verify that $\mathcal{B} = \{1, 2, 3, 4\}$ is a feasible vertex by verifying that $\mathbf{A}_\mathcal{B}$ is invertible. We have:

$$\mathbf{A}_\mathcal{B} = \begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & -2 & 0 & 1 \\ 2 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}$$

which is invertible.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We begin an iteration of the simplex algorithm by extracting $\mathbf{x}_\mathcal{B}$:

$$\mathbf{x}_\mathcal{B} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b} = \begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & -2 & 0 & 1 \\ 2 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ 1 \\ 7 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

We then compute $\boldsymbol{\mu}_\mathcal{V}$:

$$\boldsymbol{\mu}_\mathcal{V} = \mathbf{c}_\mathcal{V} - \left(\mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_\mathcal{V}\right)^\top \mathbf{c}_\mathcal{B} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \left(\begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & -2 & 0 & 1 \\ 2 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \right)^\top \begin{bmatrix} 1 \\ 3 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -4/3 \\ -5/3 \end{bmatrix}$$

We find that $\boldsymbol{\mu}_\mathcal{V}$ contains negative elements, so our current $\mathcal{B}$ is suboptimal. We can select from the two negative elements, and will choose according to the minimum ratio test. The ratios are:

$$\mathbf{x}'_{(q=5)} = \left[\frac{(\mathbf{x}_\mathcal{B})_p}{\left(\mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_{\{5\}}\right)_p} \text{ for } p \in \{1,2,3,4\} \right] = [6, 9, 6, 15]$$

$$\mathbf{x}'_{(q=6)} = \left[\frac{(\mathbf{x}_\mathcal{B})_p}{\left(\mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_{\{6\}}\right)_p} \text{ for } p \in \{1,2,3,4\} \right] = [-6, 4.5, 12, 3]$$

The minimum (positive) ratio is thus 3, achieved with $q = 6$ and $p = 4$. Hence, we update $\mathcal{B}$ by removing 4 and introducing 6 to obtain $\mathcal{B}' = \{1, 2, 3, 6\}$.

Exercise 12.5. Suppose your optimization algorithm has found a search direction $\mathbf{d}$ and you want to conduct a line search. However, you know that there is a linear constraint $\mathbf{w}^\top \mathbf{x} \geq 0$. How would you modify the line search to take this constraint into account? You can assume that your current design point is feasible.

Solution: If the current iterate $\mathbf{x}$ is feasible, then $\mathbf{w}^\top \mathbf{x} \geq 0$. We want the next point to maintain feasibility, and thus we require $\mathbf{w}^\top (\mathbf{x} + \alpha \mathbf{d}) \geq 0$. Rearranging yields:

$$\alpha \mathbf{w}^\top \mathbf{d} \geq -\mathbf{w}^\top \mathbf{x}$$

If $\mathbf{w}^\top \mathbf{d} \geq 0$, the left-hand side is nonnegative for any $\alpha \geq 0$ while the right-hand side is nonpositive, so the constraint is satisfied by any $\alpha \geq 0$ and the line search can ignore it. If $\mathbf{w}^\top \mathbf{d} < 0$, dividing by $\mathbf{w}^\top \mathbf{d}$ reverses the inequality, giving an upper bound on the step:

$$\alpha \leq -\frac{\mathbf{w}^\top \mathbf{x}}{\mathbf{w}^\top \mathbf{d}}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 12.6. Suppose we have a linear program with

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

and we perform an edge transition to move index $q = 4$ into $\mathcal{B} = \{1, 2, 3\}$. What would happen if we were to select a leaving index that does not satisfy the minimum ratio test?

Solution: The minimum ratio test produces:

$$x'_q = 1 \text{ for } p = 1$$
$$x'_q = 2 \text{ for } p = 2$$
$$x'_q = 3 \text{ for } p = 3$$

According to the minimum ratio test, we would select $p = 1$ as our leaving index. If we choose $p = 2$ instead, then our new design would have $\mathcal{B} = \{1, 3, 4\}$, which corresponds to the design:

$$\mathbf{x}_\mathcal{B} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

This design has a negative component, which is not feasible for a linear program in standard form.

Exercise 12.7. Reformulate the linear program

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{0} \end{aligned}$$

into an unconstrained optimization problem with a log barrier penalty.

Solution: We can rewrite the problem:

$$\underset{\mathbf{x}}{\text{minimize}}\, \mathbf{c}^\top \mathbf{x} - \mu \sum_i \log\left(\mathbf{A}_{\{i\}}^\top \mathbf{x}\right)$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

13 Quadratic Programming

Quadratic programming involves solving optimization problems with quadratic objective functions and linear constraints. Many problems are naturally described by quadratic programs, including problems in physics, controls, economics, and operations research. Many problems that are not naturally quadratic can often be approximated by quadratic programs.[1]

13.1 Problem Formulation

Like a linear program, a quadratic programming problem, or *quadratic program*, can be expressed in multiple forms. One general form for a quadratic program is the linear program from equation (12.2) with an additional quadratic objective term:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{q}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{C}_{\text{LE}}\mathbf{x} \leq \mathbf{d}_{\text{LE}} \\ & \mathbf{C}_{\text{EQ}}\mathbf{x} = \mathbf{d}_{\text{EQ}} \end{aligned} \tag{13.1}$$

Here $\frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ produces quadratic components $\frac{1}{2}Q_{ii}x_i^2$ and product components $(\frac{1}{2}Q_{ij} + \frac{1}{2}Q_{ji})x_i x_j$ for $i < j$.

Quadratic programs naturally arise when forming a second-order approximation of a function f about a reference point $\mathbf{x}_{\text{ref}}$ and optimizing with respect to the displacement $\mathbf{x} - \mathbf{x}_{\text{ref}}$ instead of $\mathbf{x}$. In this case, we set $\mathbf{Q} = \nabla^2 f(\mathbf{x}_{\text{ref}})$ equal to the Hessian and $\mathbf{q} = \nabla f(\mathbf{x}_{\text{ref}})$ equal to the gradient. The constant term from the Taylor expansion is typically omitted since it does not affect the minimizer. Newton's method (section 6.1) solves unconstrained optimization problems through sequential second-order approximations. Quadratic programs can therefore be viewed as Newton's method subject to linear constraints.

[1] This chapter is a short introduction to quadratic programs and one method used to solve them. Several textbooks are dedicated entirely to quadratic programs, or the closely related least-squares problems, including C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Prentice-Hall, 1974. Like linear programs, quadratic programs can also be solved using common mathematical packages like `Convex.jl` and `JuMP.jl`.

Many quadratic program formulations assume a symmetric $\mathbf{Q}$. When $\mathbf{Q}$ is not symmetric, we can replace it with $\frac{1}{2}(\mathbf{Q} + \mathbf{Q}^\top)$ without affecting the solution.[2]

[2] See exercise 13.5 for a derivation.

When $\mathbf{Q}$ is positive definite, then $\mathbf{x}^\top\mathbf{Q}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$ and the objective is strictly convex. It is then possible to factor $\mathbf{Q}$ into $\mathbf{U}^\top\mathbf{U}$ using a Cholesky decomposition (appendix C.8.1), and the quadratic program can be written as a *least-squares problem*:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \\ \text{subject to} \quad & \mathbf{C}_{\text{LE}}\mathbf{x} \leq \mathbf{d}_{\text{LE}} \\ & \mathbf{C}_{\text{EQ}}\mathbf{x} = \mathbf{d}_{\text{EQ}} \end{aligned} \tag{13.2}$$

with $\mathbf{A} = \mathbf{U}$ and $\mathbf{b} = -\mathbf{U}^{-\top}\mathbf{q}$. A solution to a least-squares problem $\mathbf{x}^*$ is a vector that minimizes the Euclidean length of $\mathbf{A}\mathbf{x} - \mathbf{b}$ while satisfying the linear constraints, as shown in example 13.1.[3]

[3] Minimizing the Euclidean length is equivalent to minimizing its square: $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$, which is a quadratic objective function. Problem equivalence is demonstrated in exercise 13.6.

This chapter will focus on quadratic programs that can be represented as least-squares problems.[4] We will approach the solution of such a quadratic program through a sequence of problem transformations (figure 13.1). The result of the final transformation is a problem we can solve directly, and the results can be propagated back to determine the solution to the original program.

[4] The least-squares problem derived from a quadratic program through the Cholesky decomposition will have a square matrix $\mathbf{A}$. However, the least-squares algorithms in this chapter support non-square matrices.

The feasible set for a quadratic program has the same form as the feasible set for a linear program, which is a convex set formed by the intersection of half-spaces defined by linear inequality constraints. However, unlike linear programs, where an optimal solution is guaranteed to occur at a vertex, the optimal solution to a quadratic program may be unique and lie either in the interior of the feasible set or in the relative interior of a face. Several such cases are shown in figure 13.2.

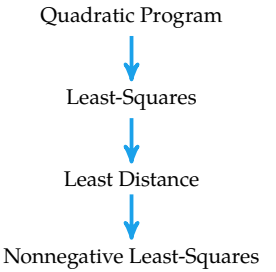


Figure 13.1. The series of transformations applied to quadratic programs with positive-definite $\mathbf{Q}$ matrices.

13.2 Unconstrained Least-Squares Problems

We first consider least-squares problems without any constraints:

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \tag{13.3}$$

The objective is minimized when $\mathbf{A}\mathbf{x}$ is as close to $\mathbf{b}$ as possible.[5] If $\mathbf{A}$ is invertible, then we have a unique solution that drives the objective to zero:

$$\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b} \tag{13.4}$$

[5] This problem is solved in Julia using the *backslash operator* by writing `x = A \ b` as discussed in appendix A.1.5. This section outlines how to solve this problem using the pseudoinverse.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 13.1. An example quadratic program. The objective function forms elliptical contours, the center of which may not lie in the feasible set. Notice that the minimum need not lie at a vertex. Here, the solution is at $\mathbf{x}^* = [1.4, 0.8]$.

The following problem has a quadratic objective and linear constraints, making it a quadratic program:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & \left\| \begin{bmatrix} 2 & 1 \\ -4 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\| \\ \text{subject to} \quad & \begin{aligned} 2x_1 \quad & -1x_2 \quad && \geq 2 \\ 1x_1 \quad & -3x_2 \quad && \leq 4 \end{aligned} \end{aligned}$$

Below we show a contour plot of the objective function. The linear constraint boundaries are shown with the feasible set shaded.

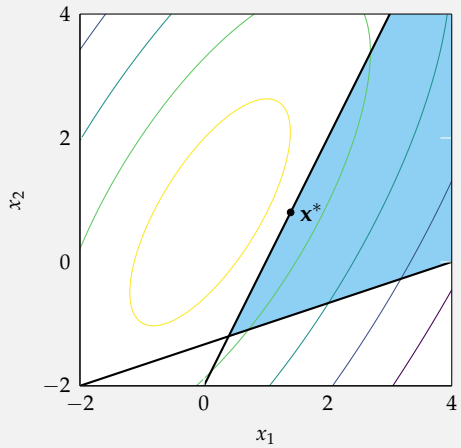
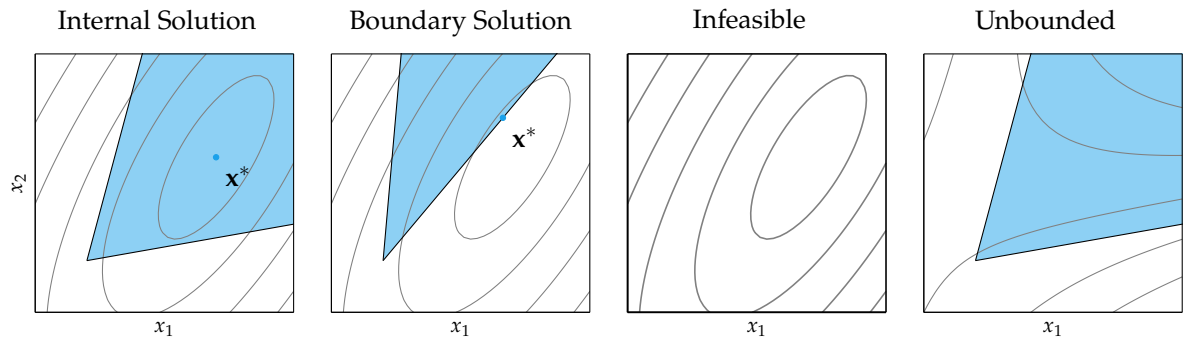


Figure 13.2. Quadratic program constraint forms with different solutions. In the last case, $\mathbf{Q}$ is not positive definite and the particular feasible set leads to an unbounded solution.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If $\mathbf{A}$ is not invertible, then solving the least-squares problem is slightly more complicated, but we can still derive a solution analytically.[6] In fact, multiple solutions may exist. An $\mathbf{A} \in \mathbb{R}^{m \times n}$ of rank k always has a *singular value decomposition* (appendix C.8.4) that can be written

$$\mathbf{A} = \mathbf{U} \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{V}^\top \tag{13.5}$$

for orthogonal matrix $\mathbf{U} \in \mathbb{R}^{m \times m}$, orthogonal matrix $\mathbf{V} \in \mathbb{R}^{n \times n}$, and diagonal matrix $\mathbf{T} \in \mathbb{R}^{k \times k}$ with positive entries. We can substitute this decomposition into the square of our objective function:[7]

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \left\| \mathbf{U} \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{V}^\top \mathbf{x} - \mathbf{b} \right\|^2 \tag{13.6}$$

An orthogonal matrix's inverse is equal to its transpose, and multiplication by an orthogonal matrix preserves Euclidean length:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \left\| \mathbf{U}^{-1}\mathbf{U} \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{V}^\top \mathbf{x} - \mathbf{U}^{-1}\mathbf{b} \right\|^2 \tag{13.7}$$

$$= \left\| \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \left(\mathbf{V}^\top \mathbf{x}\right) - \left(\mathbf{U}^\top \mathbf{b}\right) \right\|^2 \tag{13.8}$$

If we split our matrices after the first k columns, we can further simplify to:

$$\mathbf{U} = [\mathbf{U}_1, \mathbf{U}_2] \qquad \mathbf{V} = [\mathbf{V}_1, \mathbf{V}_2] \qquad \mathbf{U}_1 \in \mathbb{R}^{m \times k} \qquad \mathbf{V}_1 \in \mathbb{R}^{n \times k} \tag{13.9}$$

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \left\| \begin{bmatrix} \mathbf{T}\mathbf{V}_1^\top \mathbf{x} - \mathbf{U}_1^\top \mathbf{b} \\ -\mathbf{U}_2^\top \mathbf{b} \end{bmatrix} \right\|^2 \tag{13.10}$$

$$= \left\| \mathbf{T}\mathbf{V}_1^\top \mathbf{x} - \mathbf{U}_1^\top \mathbf{b} \right\|^2 + \left\| \mathbf{U}_2^\top \mathbf{b} \right\|^2 \tag{13.11}$$

Let us split $\mathbf{V}^\top \mathbf{x}$ into a component $\mathbf{v}_{\text{fixed}} = \mathbf{V}_1^\top \mathbf{x}$ that is active in the objective and a component $\mathbf{v}_{\text{free}} = \mathbf{V}_2^\top \mathbf{x}$ that is not. Varying the free component does not change the objective value, making it free to vary.

The objective only depends on the active component:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = \left\| \mathbf{T}\mathbf{v}_{\text{fixed}} - \mathbf{U}_1^\top \mathbf{b} \right\|^2 + \left\| \mathbf{U}_2^\top \mathbf{b} \right\|^2 \tag{13.12}$$

[6] Alternative derivations are provided in Section 12.2 of the textbook by S. Boyd and L. Vandenberghe, *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018.

[7] Minimizing the square of the Euclidean length results in the same set of minimizers as minimizing the Euclidean length because the square function is strictly increasing on the nonnegative reals. We square the objective to simplify the derivation.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and is minimized when

$$\mathbf{v}_{\text{fixed}}^* = \mathbf{T}^{-1}\mathbf{U}_1^\top \mathbf{b} \tag{13.13}$$

While $\mathbf{v}_{\text{fixed}}^*$ is unique, the overall optimal design can be nonunique because $\mathbf{v}_{\text{free}}$ can take on any value. Setting $\mathbf{v}_{\text{free}} = \mathbf{0}$ yields the unique solution of minimum Euclidean norm. In this case,

$$\begin{aligned}
\mathbf{x}^* &= \mathbf{V}\begin{bmatrix}\mathbf{v}_{\text{fixed}}^* \\ \mathbf{0}\end{bmatrix} && (13.14)\\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1}\mathbf{U}_1^\top \mathbf{b} \\ \mathbf{0}\end{bmatrix} && (13.15)\\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{U}_1^\top \\ \mathbf{U}_2^\top\end{bmatrix}\mathbf{b} && (13.16)\\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top \mathbf{b} && (13.17)
\end{aligned}$$

We can show that the equation above is equivalent to

$$\mathbf{x}^* = \mathbf{A}^+ \mathbf{b} \tag{13.18}$$

where $\mathbf{A}^+$ is the *pseudoinverse*[8] of $\mathbf{A}$. The pseudoinverse can be computed with a singular value decomposition:

$$\mathbf{A}^+ = \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top \tag{13.19}$$

In particular, $\mathbf{A}^+$ can be computed using matrix transposes, inverses, and products depending on the linear independence of the rows or columns:[9]

$$\mathbf{A}^+ = \left(\mathbf{A}^\top \mathbf{A}\right)^{-1}\mathbf{A}^\top \quad \text{when the columns of } \mathbf{A} \text{ are linearly independent} \tag{13.20}$$

$$\mathbf{A}^+ = \mathbf{A}^\top\left(\mathbf{A}\mathbf{A}^\top\right)^{-1} \quad \text{when the rows of } \mathbf{A} \text{ are linearly independent} \tag{13.21}$$

$$\mathbf{A}^+ = \mathbf{A}^{-1} \quad \text{when } \mathbf{A} \text{ is invertible} \tag{13.22}$$

[8] The function `pinv` in Julia computes the pseudoinverse of a matrix.

[9] Exercise 13.8 shows that these formulations are equivalent to equation (13.19).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

13.3 Least Squares with Linear Inequalities

The general least-squares program given in equation (13.2) can be converted into a *least squares with linear inequalities*. Here, all constraints are greater-than inequalities:[10]

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \\ \text{subject to} \quad & \mathbf{C}\mathbf{x} \geq \mathbf{d} \end{aligned} \tag{13.23}$$

[10] Use of greater-than inequalities is both standard in the literature and prepares for an eventual $\mathbf{x} \geq 0$ form in a later representation.

To arrive at this form, less-than inequalities can simply be negated. While equality constraints could be split into two, as was done with linear programs, it is more common to eliminate them as covered in section 10.4. Elimination avoids numerical instability that can occur when working with competing inequalities without an interior.

Equality constraints are eliminated using an LQ decomposition[11] $\mathbf{C}_{\text{EQ}} = \mathbf{L}\mathbf{Q}$ and replacing $\mathbf{x}$ in our problem with

$$\mathbf{x} = \mathbf{Q}^\top \begin{bmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2 \end{bmatrix} \tag{13.24}$$

where $\mathbf{y}_1^* = \mathbf{L}_{11}^{-1}\mathbf{d}_{\text{EQ}} \in \mathbb{R}^c$ and c is the number of equality constraints.[12] This substitution results in

$$\begin{aligned} \underset{\mathbf{y}_2}{\text{minimize}} \quad & \left\| \mathbf{A}\mathbf{Q}^\top \begin{bmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2 \end{bmatrix} - \mathbf{b} \right\| \\ \text{subject to} \quad & \mathbf{C}\mathbf{Q}^\top \begin{bmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2 \end{bmatrix} \geq \mathbf{d} \end{aligned} \tag{13.25}$$

which can be simplified to

$$\begin{aligned} \underset{\mathbf{y}_2}{\text{minimize}} \quad & \left\| \left(\mathbf{A}\mathbf{Q}^\top\right)_{(\cdot,c+1:n)} \mathbf{y}_2 - \left(\mathbf{b} - \left(\mathbf{A}\mathbf{Q}^\top\right)_{(\cdot,1:c)} \mathbf{y}_1^*\right) \right\| \\ \text{subject to} \quad & \left(\mathbf{C}\mathbf{Q}^\top\right)_{(\cdot,c+1:n)} \mathbf{y}_2 \geq \left(\mathbf{d} - \left(\mathbf{C}\mathbf{Q}^\top\right)_{(\cdot,1:c)} \mathbf{y}_1^*\right) \end{aligned} \tag{13.26}$$

Algorithm 13.1 implements a method for solving a quadratic program in general form by following this conversion. Example 13.2 demonstrates this algorithm.

[11] We assume that $\mathbf{C}_{\text{EQ}}$ is full-row rank, meaning no equality constraints are redundant or contradictory. If it is not, $\mathbf{L}$ will have zeros on its lower-triangular diagonal and be singular. In practice, redundant constraints are often removed as a preprocessing step.

[12] Here we have used

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{11} & \mathbf{L}_{12} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix}$$

with $\mathbf{L}_{11} \in \mathbb{R}^{c \times c}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct QuadraticProgram
    # minimize ||Ax - b||
    A    # matrix of size m×n
    b    # vector of size m
    # subject to C_LE x ≤ d_LE
    C_LE # matrix of size (num ≤ constraints)×n
    d_LE # vector of size (num ≤ constraints)
    # subject to C_EQ x = d_EQ
    C_EQ # matrix of size (num = constraints)×n
    d_EQ # vector of size (num = constraints)
end
struct LeastSquaresWithInequalities
    # minimize ||Ax - b||
    A # matrix of size m×n
    b # vector of size m
    # subject to C x ≥ d
    C # matrix of size c×n
    d # vector of size c
end
function solve(qp::QuadraticProgram)
    A, b, C_LE, d_LE = qp.A, qp.b, qp.C_LE, qp.d_LE
    C_EQ, d_EQ = qp.C_EQ, qp.d_EQ

    c, n = size(C_EQ)
    L, Q = lq(C_EQ)
    y1 = L[:,1:c] \ d_EQ

    AQ = A*Q'
    A1, A2 = AQ[:,1:c], AQ[:,c+1:end]
    CQ = -C_LE*Q'
    C1, C2 = CQ[:,1:c], CQ[:,c+1:end]
    d = -d_LE

    lsi = LeastSquaresWithInequalities(
        A2, b - A1*y1, C2, d - C1*y1)
    y2, solved = solve(lsi)
    return (Q' * [y1; y2], solved)
end
```

Algorithm 13.1. A method for solving a quadratic program in general form. This is accomplished by converting it into a least-squares problem with linear inequalities, solving that, and then backing out the original solution. This method assumes that $\mathbf{C}_{\text{EQ}} \in \mathbb{R}^{c \times n}$ has rank $c < n$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 13.2. An example showing how a three-dimensional quadratic program with an equality constraint can be transformed into a two-dimensional least-squares problem with linear inequalities. The solution is $\mathbf{x} = [0, 0.2, 0.2]$.

Consider the following quadratic program in general form:

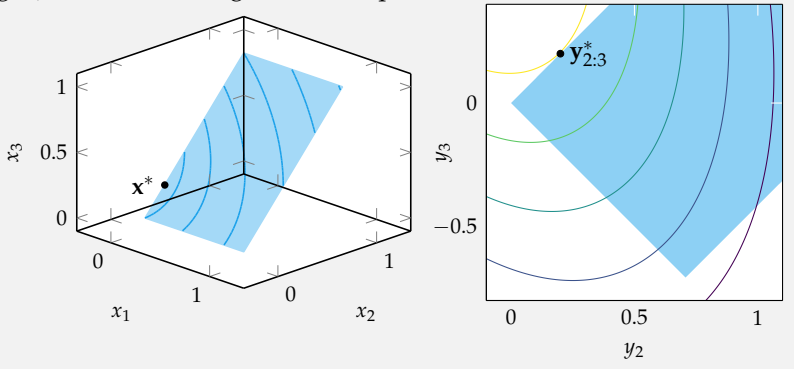
$$\begin{aligned}
\underset{\mathbf{x}}{\text{minimize}} \quad & \left\| \begin{bmatrix} 1 & 2 & 3 \\ -2 & 0 & 0 \end{bmatrix} \mathbf{x} - \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\| \\
\text{subject to} \quad & \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
& \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\
& \begin{bmatrix} 0 & -1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 0 \end{bmatrix}
\end{aligned}$$

This quadratic program is constrained to lie within a square planar region $0 \leq x_1 \leq 1, 0 \leq x_3 \leq 1$, and $x_2 = x_3$.

We can solve this quadratic program by first transforming it into the least-squares program with inequalities as given in equation (13.26) according to algorithm 13.1. When we do this, we get $\mathbf{y}_1 = [0]$ and we solve:

$$\begin{aligned}
\underset{\mathbf{y}_{2:3}}{\text{minimize}} \quad & \left\| \begin{bmatrix} 3.207 & 1.793 \\ -1.414 & 1.414 \end{bmatrix} \mathbf{y}_{2:3} - \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\| \\
\text{subject to} \quad & \begin{bmatrix} 0.500 & 0.500 \\ 0.707 & -0.707 \\ -0.500 & -0.500 \\ -0.707 & 0.707 \end{bmatrix} \mathbf{y}_{2:3} \geq \begin{bmatrix} 0 \\ 0 \\ -1 \\ -1 \end{bmatrix}
\end{aligned}$$

Below we show the original problem (left) and the simplified problem (right). The feasible region in both problems is shown in blue.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

13.4 Least Distance Programming

A least-squares problem with linear inequalities can be solved by transforming it into a *least distance program*:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x}\| \\ \text{subject to} \quad & \mathbf{G}\mathbf{x} \geq \mathbf{h} \end{aligned} \tag{13.27}$$

where $\mathbf{G} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{h} \in \mathbb{R}^m$. In other words, we have n design variables and we seek a design that is as close to the origin as possible while satisfying the inequalities $\mathbf{G}\mathbf{x} \geq \mathbf{h}$. Figure 13.3 shows a simple least distance program.

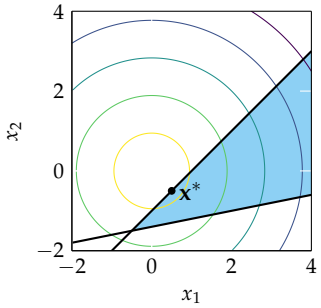


Figure 13.3. An example least distance program with

$$\mathbf{G} = \begin{bmatrix} 2.0 & -2.0 \\ -1.0 & 5.0 \end{bmatrix}$$

$$\mathbf{h} = \begin{bmatrix} 2.0 \\ -7.0 \end{bmatrix}$$

As we saw in equation (13.12), minimizing $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|$ in our least-squares problem with linear inequalities is equivalent to minimizing

$$\left\| \mathbf{T}\mathbf{v}_{\text{fixed}} - \mathbf{U}_1^\top \mathbf{b} \right\|^2 \tag{13.28}$$

where $\mathbf{v}_{\text{fixed}} = \mathbf{V}_1^\top \mathbf{x}$ and $\mathbf{U}$, $\mathbf{T}$, and $\mathbf{V}$ are given by the singular value decomposition in equation (13.5). With this transformation of the objective, the minimizers will remain the same, but the value of the minimum may change.

Using this objective and substituting $\mathbf{x} = \mathbf{V}_1 \mathbf{v}_{\text{fixed}}$ into the inequality for our least-squares problem with linear inequalities yields

$$\begin{aligned} \underset{\mathbf{v}_{\text{fixed}}}{\text{minimize}} \quad & \left\| \mathbf{T}\mathbf{v}_{\text{fixed}} - \mathbf{U}_1^\top \mathbf{b} \right\|^2 \\ \text{subject to} \quad & \mathbf{C}\mathbf{V}_1 \mathbf{v}_{\text{fixed}} \geq \mathbf{d} \end{aligned} \tag{13.29}$$

We then drop the squaring and apply another change of variables,

$$\mathbf{z} = \mathbf{T}\mathbf{v}_{\text{fixed}} - \mathbf{U}_1^\top \mathbf{b} \tag{13.30}$$

and our problem becomes a least distance program:

$$\begin{aligned} \underset{\mathbf{z}}{\text{minimize}} \quad & \|\mathbf{z}\| \\ \text{subject to} \quad & \mathbf{C}\mathbf{V}_1 \mathbf{T}^{-1} \mathbf{z} \geq \mathbf{d} - \mathbf{C}\mathbf{V}_1 \mathbf{T}^{-1} \mathbf{U}_1^\top \mathbf{b} \end{aligned} \tag{13.31}$$

Algorithm 13.2 converts a least-squares problem with inequalities into a least distance program, solves it, and backs out the solution. Figure 13.4 shows a least distance program obtained from a least-squares problem with inequalities.

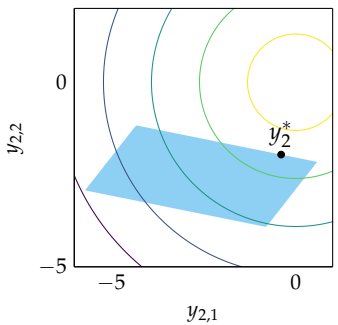


Figure 13.4. The least-squares problem with inequalities from example 13.2 can be converted to a least distance program with

$$\mathbf{G} = \begin{bmatrix} -0.176 & 0.138 \\ -0.099 & -0.49 \\ 0.176 & -0.138 \\ 0.099 & 0.49 \end{bmatrix}$$

$$\mathbf{h} = \begin{bmatrix} -0.4 \\ 1.0 \\ -0.6 \\ -2.0 \end{bmatrix}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct LeastDistanceProgram
    # minimize ||x||
    # subject to G x ≥ h
    G # matrix of size m×n
    h # vector of size m
end
function solve(qp::LeastSquaresWithInequalities)
    A, b, C, d = qp.A, qp.b, qp.C, qp.d

    res = svd(A) # thin factorization
    tol = maximum(size(A)) * eps(res.S[1])
    k = count(>(tol), res.S)
    U1 = res.U[:, 1:k]              # m × k
    Tinv = Diagonal(res.S[1:k].^-1) # k × k
    V1 = res.V[:, 1:k]              # n × k

    ldp = LeastDistanceProgram(
        C*V1*Tinv, d - C*V1*Tinv*U1'*b)
    z, solved = solve(ldp)

    return (V1*Tinv*(z + U1'*b), solved)
end
```

Algorithm 13.2. A method for solving a least-squares program with linear inequalities. This is accomplished by converting it into a least distance program, solving that, and then backing out the original solution.

We will solve a least distance program by transforming it into a *nonnegative least-squares* (NNLS) problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{E}\mathbf{x} - \mathbf{f}\| \\ \text{subject to} \quad & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{13.32}$$

where $\mathbf{E} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{f} \in \mathbb{R}^m$. In other words, we have n nonnegative design variables and we seek to minimize the Euclidean length of a system of m equations. Geometrically, we are trying to find the point in the positive quadrant closest to $\mathbf{f}$ under the linear transform $\mathbf{E}$. Figure 13.5 shows a simple nonnegative least-squares problem.

The least distance program is converted into an NNLS problem:[13]

$$\begin{aligned} \underset{\mathbf{y}}{\text{minimize}} \quad & \left\| \begin{bmatrix} \mathbf{G}^\top \\ \mathbf{h}^\top \end{bmatrix} \mathbf{y} - \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} \right\| \\ \text{subject to} \quad & \mathbf{y} \geq \mathbf{0} \end{aligned} \tag{13.33}$$

The corresponding NNLS problem thus has as many design variables as $\mathbf{G}$ has rows.

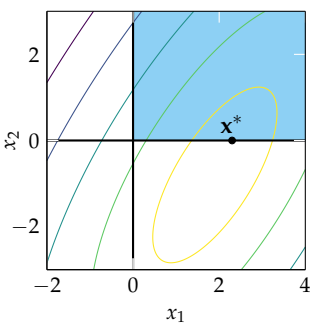


Figure 13.5. An example nonnegative least-squares quadratic program with

$$\mathbf{E} = \begin{bmatrix} 2 & 1 \\ -4 & 3 \end{bmatrix} \qquad \mathbf{f} = \begin{bmatrix} 3 \\ -10 \end{bmatrix}$$

[13] The variables $\mathbf{y}$ are proportional to the dual variables of the least distance program.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The solution $\mathbf{y}^*$ to the NNLS problem can be used to obtain the solution $\mathbf{x}^*$ to our least distance problem. First, we compute the residual vector:

$$\mathbf{r} = \mathbf{E}\mathbf{y}^* - \mathbf{f} \tag{13.34}$$

Then, we construct an n-dimensional solution vector $\mathbf{x}^*$ with components:

$$x_j^* = -\frac{r_j}{r_{n+1}} \tag{13.35}$$

for $j = 1, \ldots, n$ and $r_{n+1} < 0$.

This procedure is implemented in algorithm 13.3. It is used to solve an example least distance program in example 13.3.

```
function solve(qp::LeastDistanceProgram, ε=√eps(Float64))
    G, h = qp.G, qp.h
    m, n = size(G)
    x = zeros(n)

    E = [G'; h']
    f = [zeros(n); 1]
    y, solved = solve(NonnegativeLeastSquares(E, f))
    if !solved
        return (x, false) # failed to solve NNLS
    end

    r = E*y - f
    if r[n+1] ≥ -ε
        return (x, false) # zero residual - malformed problem
    end

    return (-r[1:n]./r[n+1], true) # success
end
```

Algorithm 13.3. A method for solving a least distance program `qp`. The `solve` method returns both the solution vector and whether a solution was found.

The solution $\mathbf{x}^*$ can only be optimal in our original problem if the KKT conditions are satisfied. To show this, we will use the fact that the squared length of our residual vector is $-r_{n+1}$. This can be shown by expanding:

$$\begin{aligned}
\|\mathbf{r}\|^2 &= \mathbf{r}^\top \mathbf{r} && (13.36)\\
&= \mathbf{r}^\top (\mathbf{E}\mathbf{y}^* - \mathbf{f}) && (13.37)\\
&= \mathbf{r}^\top \mathbf{E}\mathbf{y}^* - \mathbf{r}^\top \mathbf{f} && (13.38)\\
&= \left(\mathbf{E}^\top \mathbf{r}\right)^\top \mathbf{y}^* - \mathbf{r}^\top \mathbf{f} && (13.39)
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can show that the left term $\left(\mathbf{E}^\top \mathbf{r}\right)^\top \mathbf{y}^*$ is zero. First, we recognize that minimizing $\frac{1}{2}\|\mathbf{E}\mathbf{y} - \mathbf{f}\|^2$ is equivalent to minimizing the NNLS objective function. From the KKT conditions, we know that the gradient of an objective function at a solution is balanced by the constraints:

$$\nabla f(\mathbf{y}^*) + \sum_i \mu_i \nabla g_i(\mathbf{y}^*) = \mathbf{0} \tag{13.40}$$

$$\nabla_\mathbf{y} \frac{1}{2}\|\mathbf{E}\mathbf{y} - \mathbf{f}\|^2 + \boldsymbol{\mu}(-1) = \mathbf{0} \tag{13.41}$$

$$\mathbf{E}^\top(\mathbf{E}\mathbf{y}^* - \mathbf{f}) - \boldsymbol{\mu} = \mathbf{0} \tag{13.42}$$

$$\mathbf{E}^\top \mathbf{r} - \boldsymbol{\mu} = 0 \tag{13.43}$$

$$\boldsymbol{\mu} = \mathbf{E}^\top \mathbf{r} \tag{13.44}$$

The left term can thus be interpreted as $\boldsymbol{\mu}^\top \mathbf{y}^*$. By dual feasibility we know $\boldsymbol{\mu} \geq \mathbf{0}$ and by complementary slackness $\boldsymbol{\mu}^\top \mathbf{g}(\mathbf{y}^*) = 0$. Since $\mathbf{g}(\mathbf{y}^*) = -\mathbf{y}^*$, the term is zero.

The right term, $\mathbf{r}^\top \mathbf{f}$, is equal to r_{n+1} because the first n components of $\mathbf{f}$ in our NNLS problem are zero. Hence, $\|\mathbf{r}\|^2 = -r_{n+1}$.

Now we can show that $\mathbf{x}^*$ is a feasible solution. If we solved our NNLS problem, then the gradient of its objective function $\mathbf{E}^\top \mathbf{r}$ will be nonnegative.[14] As such:

[14] This arises from equation (13.42) and the fact that $\boldsymbol{\mu} \geq \mathbf{0}$.

$$\mathbf{0} \leq \mathbf{E}^\top \mathbf{r} = \begin{bmatrix} \mathbf{G} & \mathbf{h} \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ -1 \end{bmatrix} (-r_{n+1}) = (\mathbf{G}\mathbf{x}^* - \mathbf{h})\|\mathbf{r}\|^2 \tag{13.45}$$

We can rearrange to obtain $\mathbf{G}\mathbf{x}^* \geq \mathbf{h}$, which proves feasibility.

Next we will show that the KKT condition of stationarity holds. The gradient of our objective function at our solution is balanced by our active constraints:

$$\nabla f(\mathbf{x}^*) + \sum_i \mu_i \nabla g_i(\mathbf{x}^*) = \mathbf{0} \tag{13.46}$$

$$\nabla_\mathbf{x} \frac{1}{2}\|\mathbf{x}\|^2 \Big|_{\mathbf{x}^*} - \boldsymbol{\mu}\nabla_\mathbf{x}(\mathbf{G}\mathbf{x}^* - \mathbf{h}) = \mathbf{0} \tag{13.47}$$

$$\mathbf{x}^* - \mathbf{G}^\top \boldsymbol{\mu} = \mathbf{0} \tag{13.48}$$

$$\mathbf{x}^* = \mathbf{G}^\top \boldsymbol{\mu} \tag{13.49}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can match this with our construction for $\mathbf{x}^*$:

$$\mathbf{x}^* = -\frac{1}{r_{n+1}}\mathbf{r}_{1:n} \tag{13.50}$$

$$= -\frac{1}{r_{n+1}}(\mathbf{E}\mathbf{y}^* - \mathbf{f})_{1:n} \tag{13.51}$$

$$= -\frac{1}{r_{n+1}}\mathbf{G}^\top \mathbf{y}^* \tag{13.52}$$

$$= \mathbf{G}^\top \mathbf{y}^* \|\mathbf{r}\|^{-2} \tag{13.53}$$

We satisfy stationarity with $\boldsymbol{\mu} = \mathbf{y}^* \|\mathbf{r}\|^{-2}$. Furthermore, we satisfy both dual feasibility and complementary slackness as $\mathbf{y}_i^*$ is zero for inactive constraints and positive for active constraints. We thus have a primal-dual pair satisfying the KKT conditions for a convex problem, which is sufficient for optimality, implying $\mathbf{x}^*$ is optimal for the least-distance problem.

13.5 Solving Nonnegative Least-Squares Problems

Recall that the simplex algorithm (section 12.2) partitions the design components into two sets $\mathcal{B}$ and $\mathcal{V}$, and traverses from vertex to vertex by reassigning components between these sets. Nonnegative least-squares problems from equation (13.32) can be solved in a similar manner.

The n design values are again partitioned into two sets $\mathcal{B}$ and $\mathcal{V}$. The design values associated with indices in $\mathcal{V}$ are zero, whereas the design values associated with indices in $\mathcal{B}$ may be positive:

$$i \in \mathcal{B} \implies x_i \geq 0 \tag{13.54}$$

$$i \in \mathcal{V} \implies x_i = 0 \tag{13.55}$$

Intuitively, $\mathcal{V}$ refers to all active constraints, and $\mathcal{B}$ refers to all inactive constraints. The indices in $\mathcal{B}$ and $\mathcal{V}$ are changed as the algorithm proceeds, which corresponds to enforcing or relaxing equality constraints.

Suppose we have a partitioning of constraints between $\mathcal{B}$ and $\mathcal{V}$ that corresponds to a solution $\mathbf{x}^*$ to our nonnegative least-squares problem. Then, $\mathbf{x}_\mathcal{V} = \mathbf{0}$ and $\mathbf{x}^*$ is a solution to $\text{minimize}_\mathbf{x} \|\mathbf{E}_\mathcal{B}\mathbf{x} - \mathbf{f}\|$. The dual variables are $\mu_i \geq 0$ for all $i \in \mathcal{V}$ and $\mu_i = 0$ for all $i \in \mathcal{B}$ for which $x_i > 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

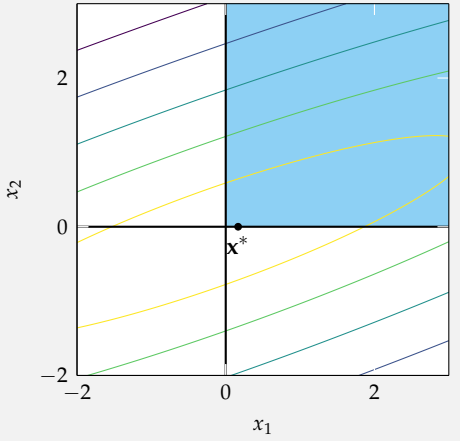
Example 13.3. Solving a simple least distance program using an intermediary nonnegative least-squares problem.

Consider the least distance program from figure 13.3 with

$$\mathbf{G} = \begin{bmatrix} 2.0 & -2.0 \\ -1.0 & 5.0 \end{bmatrix} \qquad \mathbf{h} = \begin{bmatrix} 2.0 \\ -7.0 \end{bmatrix}$$

To solve it, we first form a nonnegative least-squares problem with

$$\mathbf{E} = \begin{bmatrix} 2.0 & -1.0 \\ -2.0 & 5.0 \\ 2.0 & -7.0 \end{bmatrix} \qquad \mathbf{f} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$



The solution to the NNLS problem is $\mathbf{y}^* = [0.167, 0.000]$.

We then compute the residual, which is $\mathbf{r} = [0.333, -0.333, -0.667]$, and after verifying that it has positive norm, compute the solution to our least distance program of $\mathbf{x}^* = [0.500, -0.500]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We begin with all constraints active, $\mathcal{B} = \emptyset$, which corresponds to $\mathbf{x} = \mathbf{0}$, and compute $\boldsymbol{\mu}$ using equation (13.42). If the KKT conditions hold, then we have found a solution. Otherwise, we relax a constraint, selecting an entering index $q \in \mathcal{V}$ that will allow x_q to become positive. In our implementation, we select the index q with the most negative μ_q, and we move it from $\mathcal{V}$ to $\mathcal{B}$.

Next, we compute a new tentative solution, $\mathbf{z} = \text{minimize}_{\mathbf{x}} \|\mathbf{E}_{\mathcal{B}}\mathbf{x} - \mathbf{f}\|$. We then modify our working solution $\mathbf{x}$ by taking the largest possible step toward $\mathbf{z}$ while keeping all entries nonnegative:

$$\mathbf{x}_{\mathcal{B}} \leftarrow \mathbf{x}_{\mathcal{B}} + \alpha(\mathbf{z}_{\mathcal{B}} - \mathbf{x}_{\mathcal{B}}) \tag{13.56}$$

for $\alpha \in (0, 1]$. Here, α is usually one, but relaxing the constraint may cause other constraints to be violated, in which case at least one entry in $\mathbf{z}$ is negative. When that happens, each component j will need to satisfy $x_j + \alpha(z_j - x_j) \geq 0$. Rearranging gives us $\alpha \leq x_j/(x_j - z_j)$. We set α to the smallest $x_j/(x_j - z_j)$ among all violated entries.

Any newly constrained entries in $\mathbf{x}$ (where $x_i = 0$, or $x_i < 0$ due to numerical imprecision) are set to zero and are moved back to $\mathcal{V}$. This process is continued until all constraints are satisfied.

Algorithm 13.4 iteratively relaxes constraints until a solution is found. Note that every NNLS problem has a solution, so this algorithm will always produce an answer. This method is demonstrated in example 13.4.

13.6 Dual Certificates

Quadratic programs that satisfy Slater's condition[15] will have zero duality gap. As with section 12.3, we can verify the optimality of particular primal and dual variables by checking for primal feasibility, dual feasibility, and zero duality gap.

[15] This condition was mentioned briefly in section 11.1.

Consider a general constrained quadratic program (equation (13.1))

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{q}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{C}_{\text{LE}}\mathbf{x} \leq \mathbf{d}_{\text{LE}} \\ & \mathbf{C}_{\text{EQ}}\mathbf{x} = \mathbf{d}_{\text{EQ}} \end{aligned} \tag{13.57}$$

with positive-definite $\mathbf{Q}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct NonnegativeLeastSquares
    # minimize ||E x - f||
    # subject to x ≥ 0
    E # matrix of size m×n
    f # vector of size m
end
function solve(qp::NonnegativeLeastSquares)
    E, f = qp.E, qp.f
    m, n = size(E)
    x, μ, P, Ep = zeros(n), -E'*f, falses(n), zeros(m, n)
    q_prev = 0
    for iter in 1:3n
        if all(P) || all(μ .≥ 0.0)
            return (x, true) # success!
        end
        q = findmin(i->(!P[i] && i!=q_prev) ? μ[i] : Inf, 1:n)[2]
        q_prev = q
        P[q] = true
        Ep[:,q] = E[:,q]
        α = 0.0
        while any(P) && α != 1.0
            z = pinv(Ep) * f
            if z[q] ≤ 0.0
                μ[q] = 0.0
                break
            end
            α = minimum(z[i] < 0.0 ? x[i]/(x[i]-z[i]) : 1.0
                        for i in findall(P))
            x[P] += α * (z[P] - x[P])
            for i in 1:n
                if x[i] ≤ 0.0
                    x[i] = 0.0
                    if P[i]
                        P[i] = false
                        Ep[:,i] .= 0.0
                    end
                end
            end
        end
        μ .= E'*(E*x-f)
    end
    return (x, false) # too many iterations - malformed problem
end
```

Algorithm 13.4. A method for solving a nonnegative least-squares problem `qp`. The `solve` method returns both the solution vector and whether a solution was found. Repeatedly solving the equation $\mathbf{z} = \text{minimize}_{\mathbf{x}} \|\mathbf{E}_{\mathcal{B}}\mathbf{x} - \mathbf{f}\|$ with `pinv(Ep) * f` is computationally expensive. This algorithm can be made more efficient by maintaining a QR decomposition and applying single-column changes. For an overview and implementation, see C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Prentice-Hall, 1974.

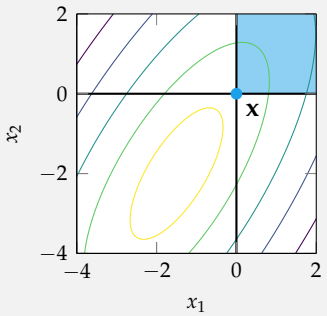
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 13.4. Algorithmic progressions on nonnegative least-squares problems.

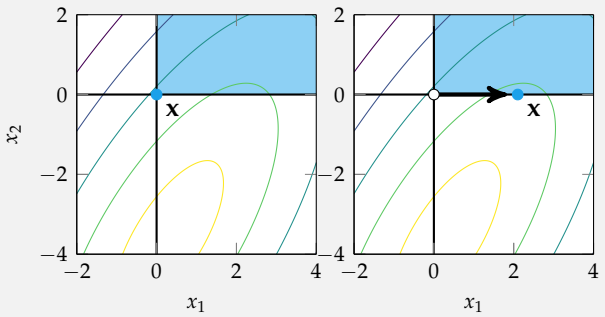
Algorithm 13.4 solves nonnegative least-squares problems by starting with all constraints active ($\mathbf{x} = \mathbf{0}$), and iteratively relaxing constraints until a solution is found. Below we show its progression on problems with

$$\mathbf{E} = \begin{bmatrix} 2.0 & 1.0 \\ -4.0 & 3.0 \end{bmatrix}$$

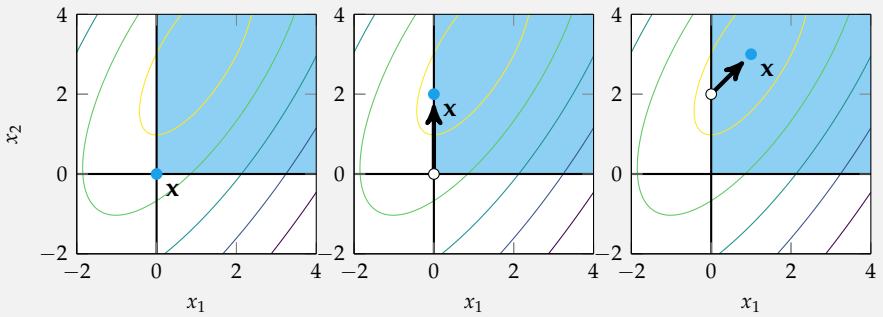
$\mathbf{f} = [-5.0, 0.0]$, solved immediately:



$\mathbf{f} = [-3.0, -12.0]$, solved after 1 relaxation:



$\mathbf{f} = [5.0, 5.0]$, solved after 2 relaxations:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Its Lagrangian is also quadratic:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{q}^\top \mathbf{x} + \boldsymbol{\mu}^\top (\mathbf{C}_{\text{LE}}\mathbf{x} - \mathbf{d}_{\text{LE}}) + \boldsymbol{\lambda}^\top (\mathbf{C}_{\text{EQ}}\mathbf{x} - \mathbf{d}_{\text{EQ}}) \tag{13.58}$$

$$= \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \left(\mathbf{q} + \mathbf{C}_{\text{LE}}^\top \boldsymbol{\mu} + \mathbf{C}_{\text{EQ}}^\top \boldsymbol{\lambda}\right)^\top \mathbf{x} + \left(-\boldsymbol{\mu}^\top \mathbf{d}_{\text{LE}} - \boldsymbol{\lambda}^\top \mathbf{d}_{\text{EQ}}\right) \tag{13.59}$$

$$= \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{e}^\top \mathbf{x} + f \tag{13.60}$$

where we choose to define $\mathbf{e} = \mathbf{q} + \mathbf{C}_{\text{LE}}^\top \boldsymbol{\mu} + \mathbf{C}_{\text{EQ}}^\top \boldsymbol{\lambda}$ and $f = -\boldsymbol{\mu}^\top \mathbf{d}_{\text{LE}} - \boldsymbol{\lambda}^\top \mathbf{d}_{\text{EQ}}$.

The dual function is

$$\mathcal{D}(\boldsymbol{\mu} \geq \mathbf{0}, \boldsymbol{\lambda}) = \operatorname*{minimize}_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{13.61}$$

$$= \operatorname*{minimize}_{\mathbf{x}} \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{e}^\top \mathbf{x} + f \tag{13.62}$$

The dual function is an unconstrained convex optimization problem, and can thus be solved by finding where the gradient is zero. We thus get an analytic solution, $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{e}$. If we make that substitution and simplify, we get:

$$\mathcal{D}(\boldsymbol{\mu} \geq \mathbf{0}, \boldsymbol{\lambda}) = -\frac{1}{2}\mathbf{e}^\top \mathbf{Q}^{-1}\mathbf{e} + f \tag{13.63}$$

The dual problem is thus:

$$\begin{aligned} \operatorname*{maximize}_{\boldsymbol{\mu}, \boldsymbol{\lambda}} \quad & -\frac{1}{2}\mathbf{e}^\top \mathbf{Q}^{-1}\mathbf{e} + f \\ \text{subject to} \quad & \boldsymbol{\mu} \geq 0 \end{aligned} \tag{13.64}$$

This dual problem may be easier to solve than the primal problem.

13.7 Summary

- Quadratic programs are optimization problems with quadratic objectives and linear constraints.
- A convex quadratic program can be equivalently written as a least-squares problem.
- Unconstrained least-squares problems can be solved exactly, using the pseudoinverse.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- A general least-squares problem can be transformed into a least-squares problem with linear inequalities.
- A least-squares problem with linear inequalities can be solved by transforming it into a least distance program, which in turn can be solved by transforming it into a nonnegative least-squares problem.
- By strong duality, the solution to the dual problem can be used to verify that a candidate primal solution is optimal.

13.8 Exercises

Exercise 13.1. Are nonnegative least-squares problems always feasible?

Solution: Yes, nonnegative least-squares problems are always feasible, as the feasible set $\mathbf{x} \geq \mathbf{0}$ always allows the positive quadrant.

Exercise 13.2. Are least distance programs always feasible?

Solution: No, least distance programs are not always feasible, as the feasible set $\mathbf{G}\mathbf{x} \geq \mathbf{h}$ may be empty. We can easily construct an empty feasible set, for example, by combining $\mathbf{1}^\top \mathbf{x} \geq 1$ and $-\mathbf{1}^\top \mathbf{x} \geq 1$.

Exercise 13.3. Do least distance programs have unique solutions?

Solution: Yes, if a least distance program has a solution, then it is unique.

Suppose that we have two optimal solutions $\mathbf{a}$ and $\mathbf{b}$ with $\|\mathbf{a}\| = \|\mathbf{b}\|$. Then we could construct $\mathbf{c} = \frac{1}{2}(\mathbf{a} + \mathbf{b})$, which would both be feasible (because our feasible set is convex) and $\mathbf{c}$ would have smaller norm. This is a contradiction, as then $\mathbf{a}$ and $\mathbf{b}$ would not be optimal. As such, solutions to LDPs are unique.

Exercise 13.4. Can all linear programs be represented as quadratic programs and least-squares problems?

Solution: Any linear program can trivially be represented as a quadratic program with $\mathbf{Q} = \mathbf{0}$.

For a linear program to be translated into a least-squares problem, we would need to be able to write a linear objective function $\mathbf{c}^\top \mathbf{x}$ in a quadratic form. We can expand the squared-norm objective:[16]

$$\begin{aligned}\mathbf{c}^\top \mathbf{x} = \frac{1}{2}\|\mathbf{A}\mathbf{x} + \mathbf{b}\|^2 &= \frac{1}{2}(\mathbf{A}\mathbf{x} + \mathbf{b})^\top (\mathbf{A}\mathbf{x} + \mathbf{b}) \\ &= \frac{1}{2}(\mathbf{A}\mathbf{x})^\top \mathbf{A}\mathbf{x} + \frac{1}{2}\mathbf{b}^\top \mathbf{A}\mathbf{x} + \frac{1}{2}(\mathbf{A}\mathbf{x})^\top \mathbf{b} + \frac{1}{2}\mathbf{b}^\top \mathbf{b} \\ &= \frac{1}{2}\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{A}\mathbf{x} + \frac{1}{2}\mathbf{b}^\top \mathbf{b}\end{aligned}$$

[16] We introduce a factor of 1/2, which does not change the minimizer.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

While we could ignore the constant $\frac{1}{2}\mathbf{b}^\top\mathbf{b}$, we would need $\mathbf{b}^\top\mathbf{A} = \mathbf{c}^\top$ and $\frac{1}{2}\mathbf{A}^\top\mathbf{A} = \mathbf{0}$. Because these constraints are not reconcilable, linear programs cannot generally be translated into least-squares problems.

Exercise 13.5. Show that we can safely assume $\mathbf{Q}$ is symmetric because if it is not, we can instead use its symmetric equivalent

$$\frac{1}{2}\left(\mathbf{Q} + \mathbf{Q}^\top\right)$$

without affecting the solution to the problem.

Solution: Any square matrix $\mathbf{Q}$ can be decomposed into symmetric and antisymmetric matrices:

$$\mathbf{Q} = \underbrace{\frac{1}{2}\left(\mathbf{Q} + \mathbf{Q}^\top\right)}_{\mathbf{Q}_{\text{sym}}} + \underbrace{\frac{1}{2}\left(\mathbf{Q} - \mathbf{Q}^\top\right)}_{\mathbf{Q}_{\text{anti}}}$$

The quadratic objective term then expands to:

$$\mathbf{x}^\top\mathbf{Q}\mathbf{x} = \mathbf{x}^\top\mathbf{Q}_{\text{sym}}\mathbf{x} + \mathbf{x}^\top\mathbf{Q}_{\text{anti}}\mathbf{x}$$

The antisymmetric term is always zero, irrespective of $\mathbf{x}$, because $\mathbf{Q}_{\text{anti}} = -\mathbf{Q}_{\text{anti}}^\top$. Replacing a square $\mathbf{Q}$ with $\mathbf{Q}_{\text{sym}}$ thus does not affect the optimization problem.

Exercise 13.6. Show that the least-squares problem in equation (13.2) is equivalent to the general form problem in equation (13.1) for a positive definite $\mathbf{Q} = \mathbf{U}^\top\mathbf{U}$.

Solution: The constraints in equations (13.1) and (13.2) are the same. To show equivalence, we can focus on the objectives:

$$\begin{aligned}
\arg\min_{\mathbf{x}} \ \|\mathbf{A}\mathbf{x} - \mathbf{b}\| &= \arg\min_{\mathbf{x}} \ \left\|\mathbf{U}\mathbf{x} + \mathbf{U}^{-\top}\mathbf{q}\right\| \\
&= \arg\min_{\mathbf{x}} \ \left\|\mathbf{U}\mathbf{x} + \mathbf{U}^{-\top}\mathbf{q}\right\|^2 \\
&= \arg\min_{\mathbf{x}} \ \left(\mathbf{U}\mathbf{x} + \mathbf{U}^{-\top}\mathbf{q}\right)^\top\left(\mathbf{U}\mathbf{x} + \mathbf{U}^{-\top}\mathbf{q}\right) \\
&= \arg\min_{\mathbf{x}} \ \left(\mathbf{x}^\top\mathbf{U}^\top + \mathbf{q}^\top\mathbf{U}^{-1}\right)\left(\mathbf{U}\mathbf{x} + \mathbf{U}^{-\top}\mathbf{q}\right) \\
&= \arg\min_{\mathbf{x}} \ \mathbf{x}^\top\mathbf{U}^\top\mathbf{U}\mathbf{x} + \mathbf{x}^\top\mathbf{U}^\top\mathbf{U}^{-\top}\mathbf{q} + \mathbf{q}^\top\mathbf{U}^{-1}\mathbf{U}\mathbf{x} + \mathbf{q}^\top\mathbf{U}^{-1}\mathbf{U}^{-\top}\mathbf{q} \\
&= \arg\min_{\mathbf{x}} \ \mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{x}^\top\mathbf{q} + \mathbf{q}^\top\mathbf{x} + \mathbf{q}^\top\mathbf{Q}^{-1}\mathbf{q} \\
&= \arg\min_{\mathbf{x}} \ \mathbf{x}^\top\mathbf{Q}\mathbf{x} + 2\mathbf{q}^\top\mathbf{x} \\
&= \arg\min_{\mathbf{x}} \ \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{q}^\top\mathbf{x}
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 13.7. What are some relative advantages and disadvantages between solving a general least-squares problem using the techniques presented in this chapter as opposed to using an interior point method?

Solution: The method presented in this chapter applies a set of transformations until the general least-squares problem is represented as a nonnegative least-squares problem. The NNLS problem is solved in an iterative manner in which constraints are iteratively relaxed until a solution is found. There are a finite number of constraints, so one can derive timing bounds on how long it will take to solve a problem of a certain size. These guarantees are perhaps the method's greatest advantage.

Interior point methods can also be used to solve quadratic programs. Interior point methods must start with a feasible point, and finding a feasible point for a general least-squares problem may be non-trivial. However, once such a point is identified, the problem can be optimized directly without needing to apply successive transformations. Interior point methods will converge to an optimum rather than arrive exactly at an optimum, but can also be stopped early and still provide a feasible solution. They are also more general, and can support problems beyond general least-squares problems.

Exercise 13.8. Use the singular value decomposition in equation (13.5) together with equations (13.20) to (13.22) to show that the pseudoinverse of a matrix is always

$$\mathbf{A}^+ = \mathbf{V} \begin{bmatrix} \mathbf{T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{U}^\top$$

Solution: When $\mathbf{A}$ is invertible, the pseudoinverse is given by equation (13.22):

$$\mathbf{A}^+ = \mathbf{A}^{-1} = \left(\mathbf{U} \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{V}^\top \right)^{-1} = \left(\mathbf{V}^\top \right)^{-1} \begin{bmatrix} \mathbf{T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}^{-1} \mathbf{U}^{-1} = \mathbf{V} \begin{bmatrix} \mathbf{T}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{U}^\top$$

where we use the fact that $\mathbf{T}$ is diagonal, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, and an orthogonal matrix's inverse is equal to its transpose.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

When the columns of $\mathbf{A}$ are linearly independent, the pseudoinverse is given by equation (13.20):

$$\begin{aligned}
\mathbf{A}^+ &= \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top \\
&= \left(\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)^\top\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)\right)^{-1}\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)^\top \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)^{-1}\left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\right) \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)^{-1}\left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\right) \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T}^2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\right)^{-1}\left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\right) \\
&= \left(\mathbf{V}^\top\right)^{-1}\begin{bmatrix}\mathbf{T}^2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}^{-1}\mathbf{V}^{-1}\left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\right) \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

When the rows of $\mathbf{A}$ are linearly independent, the pseudoinverse is given by equation (13.21):

$$\begin{aligned}
\mathbf{A}^{+} &= \mathbf{A}^{\top}\left(\mathbf{A}\mathbf{A}^{\top}\right)^{-1} \\
&= \left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^{\top}\right)^{\top}\left(\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^{\top}\right)\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^{\top}\right)^{\top}\right)^{-1} \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^{\top}\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)^{-1} \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)\left(\mathbf{U}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)^{-1} \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)\left(\mathbf{U}\begin{bmatrix}\mathbf{T}^{2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)^{-1} \\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\right)\left(\left(\mathbf{U}^{\top}\right)^{-1}\begin{bmatrix}\mathbf{T}^{2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}^{-1}(\mathbf{U})^{-1}\right) \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}\mathbf{U}\begin{bmatrix}\mathbf{T}^{-2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top} \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{T}^{-2} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top} \\
&= \mathbf{V}\begin{bmatrix}\mathbf{T}^{-1} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^{\top}
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

14 Disciplined Convex Programming

Convex optimization problems are those with a convex objective function and constraints that define a convex feasible set. Many general techniques have been developed to efficiently compute a global optimum. Unfortunately, determining whether an arbitrary nonlinear program is convex is intractable. This chapter introduces the concept of a *disciplined convex program* (*DCP*),[1] which is a type of convex optimization problem that can be written in such a way that automated methods can both verify that the problem is convex and automatically transcribe the problem into a canonical form.[2] Although not all convex problems can be framed in terms of a disciplined convex program, a wide variety of convex problems fall within this class. The canonical form can then be efficiently solved using a variety of methods.

[1] M. Grant, S. Boyd, and Y. Ye, "Disciplined Convex Programming," *Global Optimization: From Theory to Implementation*, pp. 155–210, 2006.

[2] There are several disciplined convex programming packages spanning many commonly used languages, including `Convex.jl` for Julia and `cvxpy` for Python. These packages support backend solvers such as SCS, ECOS, OSQP, and Clarabel. S. Diamond and S. Boyd, "CVXPY: A Python-Embedded Modeling Language for Convex Optimization," *Journal of Machine Learning Research*, vol. 17, pp. 1–5, 2016. A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP Solver for Embedded Systems," in *European Control Conference (ECC)*, 2013. B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An Operator Splitting Solver for Quadratic Programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020. P. J. Goulart and Y. Chen, "Clarabel: An Interior-Point Solver for Conic Programs with Quadratic Objectives," 2024. arXiv: `2405.12762`.

14.1 Canonical Form

A *convex program* is a general optimization problem in which the objective and all inequality constraints are convex, and the equality constraints are affine:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_j(\mathbf{x}) \leq 0 \qquad \text{for } j \text{ in } 1:m \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \qquad \text{for } \mathbf{A} \in \mathbb{R}^{\ell \times n},\ \mathbf{b} \in \mathbb{R}^{\ell} \end{aligned} \tag{14.1}$$

The *canonical form* of a disciplined convex program is:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} + d \\ \text{subject to} \quad & \mathbf{x} \in S \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \qquad \text{for } \mathbf{A} \in \mathbb{R}^{\ell \times n},\ \mathbf{b} \in \mathbb{R}^{\ell} \end{aligned} \tag{14.2}$$

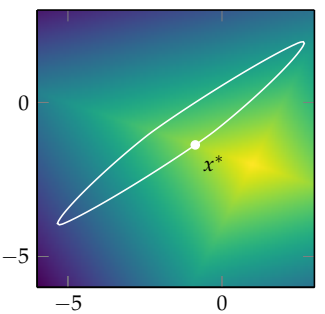
where $S \subset \mathbb{R}^n$ is a convex set. We will discuss S and the canonical form in greater detail after we have covered necessary background content. The process of disciplined convex programming converts problems into this canonical form. Example 14.1 shows an example of a disciplined convex program that can be transformed programmatically and optimized using a standard solver.

We can use `Convex.jl` to transform the following optimization problem into a canonical form that can then be solved using the SCS solver:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 \\ \text{subject to} \quad & \|\mathbf{C}\mathbf{x} - \mathbf{d}\|_{1.5} \leq 3 \end{aligned}$$

```
julia> using Convex, SCS
julia> A = [2 -1; 1 3]; b = [4, -5];
julia> C = [0 1; 3 -4]; d = [-1, 0];
julia> x = Variable(2);
julia> objective = norm(A*x - b, 1);
julia> constraint = norm(C*x - d, 1.5) ≤ 3;
julia> problem = minimize(objective, constraint);
julia> solve!(problem, SCS.Optimizer, silent=true);
julia> problem.status
OPTIMAL::TerminationStatusCode = 1
julia> problem.optval
4.356163965818378
julia> evaluate(x)
2-element Vector{Float64}:
 -0.8669274219252909
 -1.37769079355415
```

Example 14.1. Using disciplined convex programming to solve a simple optimization problem. The SCS solver was introduced by B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd, "Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding," *Journal of Optimization Theory and Applications*, vol. 169, no. 3, pp. 1042–1068, 2016.



The optimized design lies on the boundary of the feasible region, which is outlined in white.

14.2 *Verification*

We wish to have an automated way to verify whether an input problem is convex. For example, a given minimization problem is a convex program if its objective function is convex, if all of its inequality constraints are convex, and if its equality constraints are affine. Automatically verifying convex program convexity is called *verification*.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

To conduct this verification, we define a grammar.[3] Any input problem that adheres to this grammar is considered to be equivalent to a disciplined convex program, and thus a convex program. Furthermore, we can then automatically convert that input problem into canonical form.

[3] Grammars are covered in section 23.1.

The input problem can be a minimization, a maximization, or a feasibility problem.[4] All of these can be turned into minimization problems, either by negating the objective or by introducing a constant as the objective. Hence, verification can, without loss of generality, operate on minimization problems.

[4] A constrained problem with a constant objective is called a *feasibility problem* and is solved by finding any feasible design.

The constraints in a given minimization problem must be convex if they are inequality constraints, and affine if they are equality constraints. We can stipulate this more concretely by introducing a few top-level production rules for an input constraint $\mathcal{C}$:[5]

$$\begin{aligned}
\mathcal{C} &\mapsto \text{affine} = \text{affine} \\
\mathcal{C} &\mapsto \text{convex} \leq \text{concave} \\
\mathcal{C} &\mapsto \text{convex} < \text{concave} \\
\mathcal{C} &\mapsto \text{concave} \geq \text{convex} \\
\mathcal{C} &\mapsto \text{concave} > \text{convex} \\
\mathcal{C} &\mapsto (\text{affine}, \text{affine}, \ldots, \text{affine}) \in \text{convex set}
\end{aligned} \tag{14.3}$$

[5] An inequality $f(\mathbf{x}) \leq h(\mathbf{x})$ for convex f and concave h is the same as $f(\mathbf{x}) - h(\mathbf{x}) \leq 0$, where $g(\mathbf{x}) = f(\mathbf{x}) - h(\mathbf{x})$ is necessarily convex. This form matches the inequality constraints for convex programs.

These production rules ensure that all disciplined convex program constraints have one of the above forms. The final entry ensures that set membership constraints only involve affine expressions.

As we can see, determining whether an expression is affine, convex, or concave is central to disciplined convex programming. We will call this property the *curvature* of the expression. The many subsequent rules in the disciplined convex grammar will be used to determine curvature.

14.2.1 Atom Library

We need base symbols from which to construct our more complicated affine, convex, and concave expressions. Our most basic terminal symbols are numeric constants and problem variables, both of which are affine:

$$\text{affine} \mapsto \text{const} \mid \text{var} \tag{14.4}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

However, we want to compose these constants and variables with functions and sets. To this end, we define an *atom library* that encodes additional elements. We associate additional properties with each element in the atom library, most notably whether it is affine, convex, or concave. For functions, monotonicity and range are also included. We will use these properties to determine curvature when these expressions are combined.

Table 14.1 shows examples of atoms in an atom library. The last atom in that table defines x^p with a restricted domain. When we restrict the curvature of a function to a particular domain, we are effectively defining a new function where values outside of that domain are infinite. A function that can take on infinite values is called an *extended real-valued function*, or an *extended-valued function* for short. Such an extension is convenient because we do not have to include a domain qualifier. Figure 14.1 shows the extended-valued form of x^p for $p = 3$. We can use such restrictions to provide multiple definitions for a single function, capturing different curvature regions.

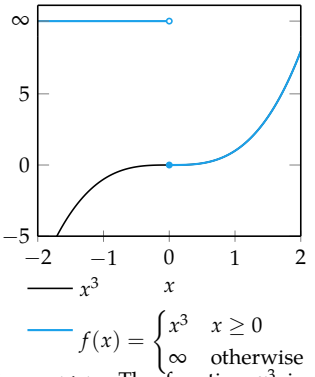


Figure 14.1. The function x^3 is not convex, but if we restrict ourselves to $x \geq 0$, then the extended-valued form is convex over all $\mathbb{R}$. Monotonicity only holds for the constrained range ($x \geq 0$).

Table 14.1. An example of atoms in an atom library.

Function	Args	Curvature	Monotonicity	Range
$\exp(x)$	`var` x	convex	increasing in x	≥ 0
$\log(x)$	`var` x	concave for $x > 0$	increasing in x	$\mathbb{R}$
x^p	`var` x, `const` p	convex for $x \geq 0$, $p > 1$	increasing in x	≥ 0

Such basic atoms give us basic forms of composition. Other forms of composition involve functions with multiple inputs. For example, we can define a more complicated expression such as $\max(\exp(x), 2 - x^3)$ by including $\max(x, y)$ as an element and allowing composition. Table 14.2 provides other examples.

Table 14.2. Examples of composition with multiple inputs.

Function	Args	Curvature	Monotonicity	Range
$x + y$	`var` x, `var` y	affine	increasing in x and y	$\mathbb{R}$
$x - y$	`var` x, `var` y	affine	increasing in x, decreasing in y	$\mathbb{R}$
$\max(x, y)$	`var` x, `var` y	convex	increasing in x and y	$\mathbb{R}$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The atom library also contains definitions for sets. Sets can be marked as convex or nonconvex. Table 14.3 provides examples.

Table 14.3. Examples of set definitions.

Set	Args	Convex	Definition
real numbers	`var x`	yes	$\{\mathbf{x} \in \mathbb{R}^n\}$
positive orthant	`var x`	yes	$\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{x} \geq 0\}$
positive definite	`var X`	yes	$\{\mathbf{X} \in \mathbb{R}^{n \times n} \mid \mathbf{u}^\top \mathbf{X}\mathbf{u} > 0 \text{ for all } \mathbf{u} \neq \mathbf{0}\}$

The atom library is extensible. Domain experts are free to introduce new functions and sets, and in so doing, enhance the capabilities of the disciplined convex solver.

14.2.2 *Product-Free Rules*

Disciplined convex programming does not allow products between non-constant expressions. This is because, in general, knowing the affine / convex / concave nature of each of the two expressions involved in a product is not enough to know whether the overall expression is affine, convex, or concave.[6] All numeric expressions must be *product-free expressions*:[7]

$$a + b_1 x_1 + b_2 x_2 + \cdots + b_n x_n + c_1 f_1(\epsilon_1) + c_2 f_2(\epsilon_2) + \ldots + c_q f_q(\epsilon_q) \tag{14.5}$$

where $\mathbf{x}$ is the design vector; a, $\mathbf{b}$, and $\mathbf{c}$ are constants; the functions $f_{1:q}$ are defined in the atom library; and each ϵ_j represents function arguments. All arguments to atom-library functions must also be product-free expressions.

For sets, unions can create ambiguity over whether the resulting set is affine, convex, concave, or otherwise. A similar set of rules exists for set expressions that constrains valid set expressions to a union-free representation.

[6] For example, $f(x_1, x_2) = x_1 x_2$ is affine in either x_1 or x_2, but it is not jointly convex in x_1 and x_2.

[7] Some examples include constants, for which $\mathbf{b} = \mathbf{0}$ and $q = 0$; affine expressions $a + \mathbf{b}^\top \mathbf{x}$, for which $q = 0$, and basic function calls, such as $\log(x_2)$, where $a = 0$, $\mathbf{b} = \mathbf{0}$, and $q = 1$. An example concave function that cannot be directly expressed as a product-free expression is $-x \log x$ for $x > 0$, though one can get around this by defining $\text{entropy}(x) = -x \log x$ in the atom library and using that instead.

14.2.3 *Sign Rules*

We can now define rules that establish whether a product-free expression is convex, concave, affine, or none of the above. We know that the $a + \mathbf{b}^\top \mathbf{x}$ portion of any product-free expression is affine, so the curvature will ultimately depend on the functions $\mathbf{f}_{1:q}$ and the constants $\mathbf{c}$. These rules are called the *sign* rules because they stipulate the signs of the $\mathbf{c}$ constants.

The sign rules state that for $c_j f_j(\cdot)$ to be a convex expression:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- If $f_j(\cdot)$ is convex, then c_j must be nonnegative.
- If $f_j(\cdot)$ is concave, then c_j must be nonpositive.
- If $f_j(\cdot)$ is affine, then c_j can have any sign.

Similarly, for $c_j f_j(\cdot)$ to be a concave expression, c_j must have the opposite sign from what it would for a convex expression. An affine expression requires that each $f_j(\cdot)$ and all of its arguments be affine. A constant expression must have $\mathbf{b} = \mathbf{0}$ and an empty $\mathbf{f}$.

Example 14.2. An application of the sign rules to a simple convex optimization problem.

Consider the following convex optimization problem:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & c_1 \exp(x_1 + 3x_2) \\ \text{subject to} \quad & 3 + 2x_1 + c_2 \max(x_1, x_2) \leq c_3 \log(x_2) \end{aligned}$$

We can verify that our objective function is convex. We have $c_1 \exp(x_1 + 3x_2)$, which involves a call to exp in our atom library. We know from our atom library that exp is convex, and so our sign rule requires that $c_1 \geq 0$. Finally, we verify that the argument is affine, which $x_1 + 3x_2$ is.

We next check our constraint, and we see that it must obey the form convex $\leq$ concave. Consequently, $3 + 2x_1 + c_2 \max(x_1, x_2)$ must be convex, which involves a call to max in our atom library. We look up max, and see that it is convex, and so our sign rule requires that $c_2 \geq 0$.

Finally, $c_3 \log(x_2)$ must be concave, which involves a call to log in our atom library. We look up log and see that it is concave, and so our sign rule requires that $c_3 \geq 0$.

14.2.4 *Composition Rules*

We now turn to enforcing the curvature of composed expressions, or expressions with function arguments that are themselves dependent on functions from the atom library, such as $f(g(x))$.

Suppose f and g are extended-valued single-argument functions.[8]

[8] We extend convex functions to $+\infty$ and concave functions to $-\infty$.

- If f is convex and nondecreasing over the range of g, and g is convex, then $f(g(x))$ is convex.

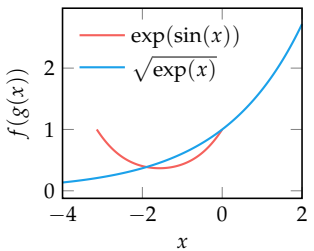
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- If f is convex and nonincreasing over the range of g, and g is concave, then $f(g(x))$ is convex.
- If f is concave and nondecreasing over the range of g, and g is concave, then $f(g(x))$ is concave.
- If f is concave and nonincreasing over the range of g, and g is convex, then $f(g(x))$ is concave.

Example 14.3. Applying the composition rules to example expressions.

Consider the expression $\exp(\sin(x))$ for $x \in [-\pi, 0]$. In this case, we have $f(g(x))$ where $f(x) = \exp(x)$ and $g(x) = \sin(x)$. We know that $\sin(x)$ is convex over $[-\pi, 0]$, with a range of $[-1, 0]$. We also know that $\exp x$ is convex and nondecreasing for all x, so the composition rules state that $\exp(\sin x)$ is convex for $x \in [-\pi, 0]$.

Next consider the expression $\sqrt{\exp(x)}$. In this case, we have $f(g(x))$ where $f(x) = \sqrt{x}$ and $g(x) = \exp(x)$. We know that $\exp(x)$ is convex, with a range of $[0, \infty]$. We also know that $\sqrt{x}$ is concave and nondecreasing for $x \in [0, \infty]$, so the composition rules are inconclusive. In fact, $\sqrt{\exp(x)}$ is convex because its second derivative $\exp(x/2)/4$ is a positive function. We cannot use the composition rules to determine the curvature of $\sqrt{\exp(x)}$. Such an expression is invalid in disciplined convex programming.



Applying these rules requires ensuring that the range of g is contained within the domain of f in which the curvature of f holds. Since we restrict ourselves to cases where g is a product-free expression, we can produce a conservative bound on its range by accumulating all of the ranges of each design variable and each constituent function:[9]

$$\text{range}(g) \subseteq a + b_1 \text{range}(x_1) + \cdots + b_n \text{range}(x_n) + c_1 \text{range}(f_1) + \cdots + c_q \text{range}(f_q) \tag{14.6}$$

where $\text{range}(f)$ provides simple interval bounds[10] for a function f and $+$ denotes Minkowski addition.[11]

This process holds for functions with multiple inputs, $f(\epsilon_1, \ldots, \epsilon_m)$. We use equation (14.6) to construct a bounded range $\mathcal{R}_i$ for each argument ϵ_i. The domain of f is therefore $(\epsilon_1, \ldots, \epsilon_m) \in \mathcal{R} = \mathcal{R}_1 \times \cdots \times \mathcal{R}_m$. Then, exactly one of the following holds:

[9] These ranges and domains of constituent functions are stored in the atom library metadata.

[10] If x is bounded to the interval $[a, b]$ and y is bounded to the interval $[c, d]$, then $x + y$ is bounded to be within the interval $[a + c, b + d]$.

[11] If A and B are sets of vectors, then the Minkowski sum of these two sets is given by $A + B = \{\mathbf{a} + \mathbf{b} \mid \mathbf{a} \in A, \mathbf{b} \in B\}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 14.4. Applying the composition rules to a more complicated expression.

Consider an $f(g(\mathbf{x}))$ expression with

$$f(x) = 1/x$$
$$g(\mathbf{x}) = 3 + 2\log(1 + x_1) - \sin x_2$$

for $x_1 \in [0, \infty]$ and $x_2 \in [-\pi, 0]$. We know that $f(x)$ is convex and nonincreasing for $x > 0$, and concave and nonincreasing for $x < 0$. Thus, the range of $g(\mathbf{x})$ is crucial in determining the curvature of our expression.

We know that $g(\mathbf{x})$ is concave, because $2 \log x$ for $x \geq 1$ is concave and $\sin x$ for $x \in [-\pi, 0]$ is convex, and so $-\sin x$ is concave. We apply equation (14.6) to our $g(\mathbf{x})$ expression to bound its range:

$$\begin{aligned}\text{range}(g) &\subseteq 3 + 2 \cdot \text{range}(\log(1 + x_1)) - \sin x_2 \\ &= [3, 3] + 2 \cdot [0, \infty] - [-1, 0] \\ &= [3, 3] + [0, \infty] + [0, 1] \\ &= [3, \infty]\end{aligned}$$

The bounded range is thus $g(\mathbf{x}) \in [3, \infty]$, a domain within which $f(x)$ is convex and nonincreasing. This fact, and the fact that $g(\mathbf{x})$ is concave, means $f(g(\mathbf{x}))$ is convex.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- If $f(\epsilon_1, \epsilon_2, \ldots, \epsilon_m)$ is expected to be convex, then f must be affine or convex, and for each argument ϵ_i:
  - f is nondecreasing in ϵ_i over $\mathcal{R}$, and ϵ_i is convex, or
  - f is nonincreasing in ϵ_i over $\mathcal{R}$, and ϵ_i is concave, or
  - ϵ_i is affine
- If $f(\epsilon_1, \epsilon_2, \ldots, \epsilon_m)$ is expected to be concave, then f must be affine or concave, and for each argument ϵ_i:
  - f is nondecreasing in ϵ_i over $\mathcal{R}$, and ϵ_i is concave, or
  - f is nonincreasing in ϵ_i over $\mathcal{R}$, and ϵ_i is convex, or
  - ϵ_i is affine
- If $f(\epsilon_1, \epsilon_2, \ldots, \epsilon_m)$ is expected to be affine, then f must be affine, and all arguments must be affine.

Examples for single and multi-input expressions are given in example 14.4 and example 14.5.

The composition rules guarantee that efficient decompositions are possible. For example, if the first rule applies and f is convex and nondecreasing, then we can decompose the inequality $f(g(x)) \leq y$ into two inequalities involving a new variable z:

$$f(g(x)) \leq y \qquad \iff \qquad \begin{aligned} f(z) &\leq y \\ g(x) &\leq z \end{aligned} \tag{14.7}$$

These decompositions are later used to partition our problem.

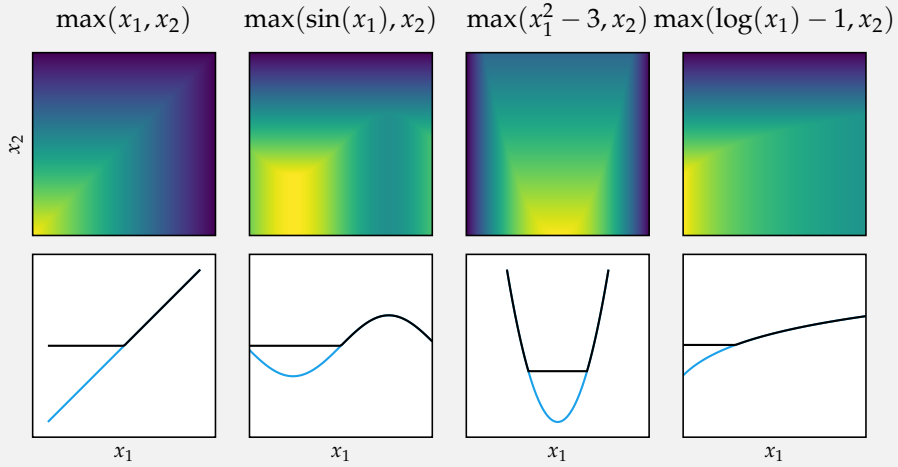
14.2.5 *Automatic Verification*

Automated verification takes a given input optimization problem and verifies that it adheres to the disciplined convex programming requirements. This verification process is typically done in two stages. The first stage verifies that all expressions are product-free. The second stage then verifies the top-level, sign, and composition rules.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 14.5. Applying the composition rules to an expression with multiple inputs. The top row of plots shows $\max(\epsilon(x_1), x_2)$. The bottom row shows $\max(\epsilon(x_1), 0)$ in black and $\epsilon(x_1)$ in blue.

Consider max, a convex function with two arguments. Below we show $\max(\epsilon(x_1), x_2)$ for several forms of ϵ:



We know that max is increasing in both arguments, and here the second argument is affine in all cases. In the case of $\epsilon_1 = x_1$, both arguments are affine and the domain of our expression is $(x_1, x_2) \in \mathbb{R}^1 \times \mathbb{R}^1 = \mathbb{R}^2$. For $\epsilon_1 = \sin(x_1)$, our argument is neither convex nor concave, so the overall expression is not convex. For $\epsilon_1 = x_1^2 - 3$, our argument is convex, so the overall expression is convex with domain $(x_1, x_2) \in \mathbb{R}^1 \times \mathbb{R}^1 = \mathbb{R}^2$. For $\epsilon_1 = \log(x_1) - 1$ for $x_1 > 0$, our argument is concave, so the overall expression is not convex.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Verifying that an expression is product-free is a recursive process. Given an input expression, we first check whether it can be written in the form of equation (14.5). Then, after verifying that all called functions are registered in the atom library, we recurse into each argument and verify that it is similarly a product-free expression. This process is shown in figure 14.2.

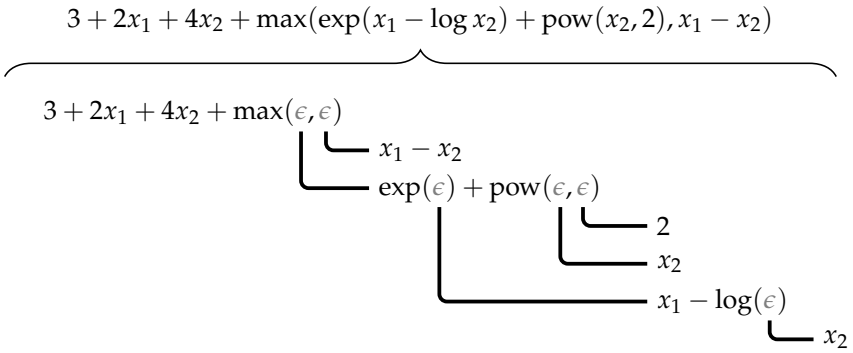


Figure 14.2. The recursive nature of product-free expressions allows them to be written as expression trees where each node is a product-free expression. The verification process produces such a tree. A demonstration of this process can be found at: `https://dcp.stanford.edu`. Note that $\mathrm{pow}(x, p)$ is the power function x^p.

The second stage in automatic verification verifies that the top-level, sign, and composition rules hold. The top-level rules determine the expected curvature for the objective function and the expressions on either side of constraints. The sign rules and composition rules can then be applied as one works down the problem's expression tree. Example 14.6 shows an instance of this process.

Packages like `Convex.jl` will perform this automatic verification. Example 14.7 presents the results from running on the problem from example 14.1.

14.3 Canonicalization

Canonicalization is the process of finding a disciplined convex program in a canonical form for an original problem specification. A collection of problem transformations can be used to produce a DCP in canonical form. These transformations additionally allow for dual information to be maintained. This information can help produce lower bounds, provide dual certificates, and assess convergence.

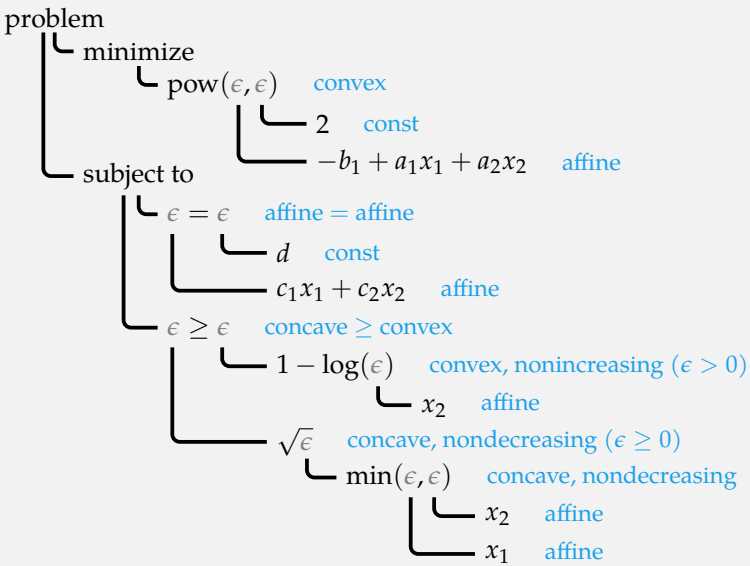
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 14.6. An illustration of automatic verification for disciplined convex programming.

Consider the following optimization problem:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & \text{pow}(a_1x_1 + a_2x_2 - b_1, 2) \\ \text{subject to} \quad & c_1x_1 + c_2x_2 = d \\ & \sqrt{\min(x_1, x_2)} \geq 1 - \log(x_2) \end{aligned}$$

In the first stage of verification, we construct expression trees for the objective and each side of each constraint and verify that these expressions are product-free. In the second stage of verification, we construct an expression tree for the overall problem, and verify the additional DCP rules.



Note that this process can be used to discover missing conditions for the problem to be a valid disciplined convex program. In this case, we require that $x_2 > 0$ in order for $\log(x_2)$ to be valid, and that $\min(x_1, x_2) \geq 0$ in order for the square root to be valid. Hence, we need $x_1 \geq 0$ and $x_2 > 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can use `Convex.jl` to show the results of automatic verification on the problem from example 14.1:

```
julia> A = [2 -1; 1 3]; b = [4, -5];
julia> C = [0 1; 3 -4]; d = [-1, 0];
julia> x = Variable(2);
julia> objective = norm(A*x - b, 1);
julia> constraint = norm(C*x - d, 1.5) ≤ 3;
julia> problem = minimize(objective, constraint)
Problem statistics
  problem is DCP         : true
  number of variables    : 1 (2 scalar elements)
  number of constraints  : 1 (1 scalar elements)
  number of coefficients : 13
  number of atoms        : 8
Solution summary
  termination status : OPTIMIZE_NOT_CALLED
  primal status      : NO_SOLUTION
  dual status        : NO_SOLUTION
Expression graph
  minimize
   └─ sum (convex; positive)
      └─ abs (convex; positive)
         └─ + (affine; real)
            ├─ …
            └─ …
  subject to
   └─ ≤ constraint (convex)
      └─ + (convex; real)
         ├─ rationalnorm (convex; positive)
         │  └─ …
         └─ [-3;;]
```

Example 14.7. Using `Convex.jl` to perform automatic verification.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Canonicalization will produce a problem in *partitioned canonical form*:

$$\begin{aligned} \underset{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(k)}}{\text{minimize}} \quad & d + \sum_{i=1}^{k} (\mathbf{c}^{(i)})^\top \mathbf{x}^{(i)} \\ \text{subject to} \quad & \mathbf{x}^{(i)} \in S^{(i)} \qquad \text{for } i \text{ in } 1:k \\ & \sum_{i=1}^{k} \mathbf{A}^{(i)} \mathbf{x}^{(i)} = \mathbf{b} \end{aligned} \tag{14.8}$$

where each set $S^{(i)}$ is a convex set defined in the atom library[12]. These sets contain any nonlinearities necessary for representing the optimization problem. Each set operates directly on a subset of $\mathbf{x}$, so any coupling happens through $\mathbf{A}\mathbf{x} = \mathbf{b}$.

[12] This includes standard sets like $\mathbb{R}^n$ and the positive orthant. As we will see later, this also includes epigraphs or hypographs of functions in the atom library.

14.3.1 *Linearization*

Linearization makes all of our product-free expressions linear and makes all function arguments affine. This process introduces many new variables and constraints, but individual expressions and constraints are simplified.

Linearization operates on the problem's product-free expressions. These expressions can either be objective functions, in which case they are standalone, or the left or right-hand sides of a constraint. Given a product-free expression:

$$a + \mathbf{b}^\top \mathbf{x} + \sum_{j=1}^{q} c_j f_j(\boldsymbol{\epsilon}_j) \tag{14.9}$$

the linearized form is:

$$a + \mathbf{b}^\top \mathbf{x} + \mathbf{c}^\top \mathbf{v} \tag{14.10}$$

with:

$$\begin{aligned} & f_j(\boldsymbol{\epsilon}_j) \leq v_j \text{ if } f_j \text{ is convex} \\ & f_j(\boldsymbol{\epsilon}_j) \geq v_j \text{ if } f_j \text{ is concave} \\ & \text{for } j \text{ in } 1:q \end{aligned} \tag{14.11}$$

where we have introduced q atom variables $\mathbf{v}$ and q new inequality constraints.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We must simplify any arguments that are not already simple variables. If f_j takes in $\boldsymbol{\epsilon}_j = \{\epsilon_{j,1}, \epsilon_{j,2}, \ldots\}$, then each input $\epsilon_{j,a}$ that is not already a simple variable is replaced by a new variable $u_{j,a}$ and bound to it with an equality constraint $u_{j,a} = \epsilon_{j,a}$.[13]

[13] For example, if we have $f_j = \log(5x + 3)$, we replace the argument via $f_j = \log(u_{j,1})$ and introduce $u_{j,1} = 5x + 3$.

When linearizing a product-free expression in an inequality constraint, we introduce a slack variable $s \geq 0$ to transform the inequality into an equality:

$$\begin{aligned} a + \mathbf{b}^\top \mathbf{x} + \mathbf{c}^\top \mathbf{v} \leq d \quad &\Rightarrow \quad a + \mathbf{b}^\top \mathbf{x} + \mathbf{c}^\top \mathbf{v} + s = d \\ a + \mathbf{b}^\top \mathbf{x} + \mathbf{c}^\top \mathbf{v} \geq d \quad &\Rightarrow \quad a + \mathbf{b}^\top \mathbf{x} + \mathbf{c}^\top \mathbf{v} - s = d \end{aligned} \tag{14.12}$$

This process is applied recursively until all expressions have been linearized. In practice, it can be simpler to work bottom up, starting with the innermost function arguments and working our way up to the top level inequalities and objective functions.

We can see how linearization produces a problem closer to the partitioned canonical form in equation (14.8). Any objective function has been linearized, with any nonlinearities moving into the constraints. Our variables are all either free, constrained to the positive orthant (like s), tied with an equality constraint (like u), or tied with an inequality constraint (like v).[14] Solving an optimization problem with the linear forms is equivalent to solving one with the original product-free expressions.

[14] The slack variable constraints $s \geq 0$ are equivalent to $s \in \mathbb{R}^+$, and thus satisfy the convex set constraint form $\mathbf{x}^{(i)} \in S^{(i)}$. Similarly, the inequality constraints $f_j(\boldsymbol{\epsilon}_j) \leq v_j$ and $f_j(\boldsymbol{\epsilon}_j) \geq v_j$ can also be viewed as convex set membership constraints for the pairs $(\boldsymbol{\epsilon}_j, v_j)$. This is why the function arguments must all be converted to simple variables.

An example of product-free expression linearization is given in example 14.8.

Linearization produces a much larger problem, but individual expressions are vastly simplified. Though there are more variables and constraints, solving these problems is no more difficult than solving the original problem, because solvers can take their sparse structure into account. Furthermore, the linear forms allow the next operation, graph expansion, to be applied, which elegantly converts problems into more solvable forms.[15]

[15] There is an additional step, set decomposition, which converts all set membership constraints into equality constraints and atomic nonlinearities. It is similar to linearization, and so is not covered in detail here.

14.3.2 *Graph Expansion*

The linearized form may not yet be directly solvable by a convex solver. Most convex solvers operate on specific forms, such as linear programs, quadratic programs, or second order cone programs.[16] Atoms in our original problem may not be supported by these solvers. Fortunately, it is often possible to transform atoms into other forms with equivalent solutions that are amenable to being solved by convex solvers.[17]

[16] *Second order cone programs* are convex optimization problems with linear objectives $\mathbf{q}^\top \mathbf{x}$, affine equality constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$, and second order cone constraints:

$$\|\mathbf{C}_i \mathbf{x} + \mathbf{d}_i\|_2 \leq \mathbf{e}_i^\top \mathbf{x} + f_i$$

[17] M. Grant and S. Boyd, "Graph Implementations for Nonsmooth Convex Programs," in *Recent Advances in Learning and Control*, 2008.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 14.8. Linearizing a product-free expression that has already been verified to obey the disciplined convex programming rules.

Consider again the product-free expression for $x_1 \in [0, \infty]$ and $x_2 \in [-\pi, 0]$:

$$\text{pow}(3 + 2\log(1 + x_1) - \sin(x_2), -1)$$

We first consider the innermost arguments, $1 + x_1$ and x_2. These are affine, and thus need not be linearized.

The next larger expression $3 + 2\log(1 + x_1) - \sin(x_2)$ is not affine. We introduce new variables and constraints, producing:

$$\begin{aligned} & 3 + 2v_1 - v_2 \\ \text{subject to} \quad & \log(u_{1,1}) \geq v_1 \\ & u_{1,1} = 1 + x_1 \\ & \sin(x_2) \leq v_2 \end{aligned}$$

We return to the top-level expression, now $\text{pow}(3 + 2v_1 - v_2, -1)$. We similarly introduce variables and constraints:

$$\begin{aligned} & v_3 \\ \text{subject to} \quad & \text{pow}(u_{3,1}, -1) \leq v_3 \\ & u_{3,1} = 3 + 2v_1 - v_2 \end{aligned}$$

We have thus successfully linearized our expression by introducing 5 new variables and 5 new constraints.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Suppose we have a disciplined convex program that contains $|x|$. This atom is nonlinear, and does not fit the form accepted by conventional linear or quadratic solvers. However, $|x|$ can be represented as the solution to a linear program:

$$|x| \quad \Rightarrow \quad \begin{array}{lll} \min\limits_{y} & y & \\ \text{subject to} & y & \geq x \\ & y & \geq -x \end{array} \tag{14.13}$$

A linearized disciplined convex program can incorporate this transformation. For example, if the original problem has a constraint $|2x + 3| \leq 5$, then its linearized form would have a graph expansion as follows:

$$\begin{array}{l} \underset{\mathbf{x}}{\text{minimize}} \quad \cdots \\ \qquad \vdots \\ \qquad v = |u| \\ \qquad u = 2x + 3 \\ \qquad v + s = 5 \\ \qquad s \geq 0 \end{array} \quad \Rightarrow \quad \begin{array}{l} \underset{\mathbf{x}, v}{\text{minimize}} \quad \cdots \\ \qquad \vdots \\ \qquad v \geq u \\ \qquad v \geq -u \\ \qquad u = 2x + 3 \\ \qquad v + s = 5 \\ \qquad s \geq 0 \end{array} \tag{14.14}$$

Replacing $|x|$ in this way has produced a linear program, and in so doing has removed the problematic nondifferentiable point. This transformation for $|x|$ can be derived using properties of convex functions.

A function is convex if and only if its *epigraph* is a convex set, and a function is concave if and only if its *hypograph* is a convex set. The epigraph is the region above and including a function, whereas the hypograph is the region below and including a function:

$$\begin{aligned} \texttt{epi} f &= \{(\mathbf{x}, y) \in \mathbb{R}^n \times \mathbb{R} \mid y \geq f(\mathbf{x})\} \\ \texttt{hypo} f &= \{(\mathbf{x}, y) \in \mathbb{R}^n \times \mathbb{R} \mid y \leq f(\mathbf{x})\} \end{aligned} \tag{14.15}$$

Examples of such sets are given in figure 14.3.

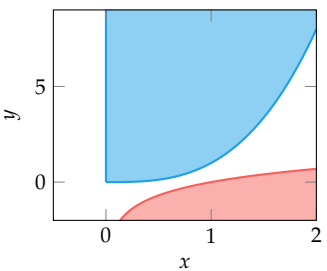


Figure 14.3. The blue region above the extended-valued convex function $f(x) = x^3$ for $x \geq 0$ is that function's epigraph. The red region below $f(x) = \log x$ for $x \geq 0$ is that function's hypograph.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can thus replace a convex or concave function with a minimization or maximization problem based on its epigraph or hypograph, respectively:

$$\begin{aligned} \text{for convex } f, \quad f(\mathbf{x}) = & \min_{y} && y \\ & \text{subject to} && (\mathbf{x}, y) \in \texttt{epi} f \end{aligned} \qquad \begin{aligned} \text{for concave } f, \quad f(\mathbf{x}) = & \max_{y} && y \\ & \text{subject to} && (\mathbf{x}, y) \in \texttt{hypo} f \end{aligned} \tag{14.16}$$

The epigraph of the real absolute value function $f(x) = |x|$ is:

$$\texttt{epi}|x| = \{(x, y) \mid y \geq x, y \geq -x\} \tag{14.17}$$

This epigraph is shown in figure 14.4.

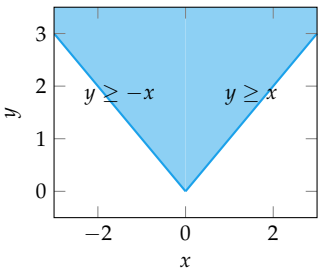


Figure 14.4. The epigraph for $|x|$ can be formed by two linear inequalities. By minimizing y we recover the original function.

Every atom in the library can have two types of implementations: a traditional implementation that provides an evaluation function, its derivatives, Hessian, etc., or a *graph implementation*, which describes the function as the solution to another DCP, typically in the standard form expected by a target solver. This process of replacing atoms with their graph implementations is called *graph expansion*.[18]

Many norms have convenient graph implementations. The norms in example 14.9 are presented to motivate disciplined convex programming. In some cases, we can recognize certain common optimization problem forms and apply similar transformations, such as the L_2 norm in example 14.10.

Disciplined convex programming allows us to automatically select appropriate transforms to efficiently solve problems. While the user could transform the input problem themselves, they may be unaware of these techniques or may find it inconvenient to transform their problems by hand. The problems shown here are relatively simple, and it can be harder for a practitioner to recognize when to tailor a problem themselves. Graph implementations let users provide their problem in the most natural form relevant to them (e.g., with an L_p-norm), allowing for a solution to be found efficiently.

14.4 Solving

Once we have a problem in partitioned canonical form, and we have applied all necessary graph expansions to convert our disciplined convex program into a form required by our targeted solver, we can then pass our problem to the solver[19]

[18] The "graph" in both names comes from hypograph and epigraph.

[19] We can apply techniques such as scaling the design variables to improve a solver's reliability. For an overview of scaling, see A. D. Belegundu and T. R. Chandrupatla, *Optimization Concepts and Applications in Engineering*, 2nd ed. Cambridge University Press, 2011.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 14.9. Graph implementations for several norms. This example was adapted from M. Grant, S. Boyd, and Y. Ye, "Disciplined Convex Programming," *Global Optimization: From Theory to Implementation*, pp. 155–210, 2006.

Both L_1 (Manhattan) norms and L_∞ (Chebyshev) norms can be converted into equivalent linear programs:

$$\|\mathbf{x}\|_1 \quad \Rightarrow \quad \begin{aligned} \underset{\mathbf{v}}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{v} \\ \text{subject to} \quad & -\mathbf{v} \leq \mathbf{x} \leq \mathbf{v} \end{aligned}$$

$$\|\mathbf{x}\|_\infty \quad \Rightarrow \quad \begin{aligned} \underset{v}{\text{minimize}} \quad & v \\ \text{subject to} \quad & -v\mathbf{1} \leq \mathbf{x} \leq v\mathbf{1} \end{aligned}$$

Again, such graph expansions change our original problem into a well-known form for which many solvers exist.

The more general L_p-norm for $p \geq 1$ can also be transformed into another DCP:

$$\|\mathbf{x}\|_p \quad \Rightarrow \quad \begin{aligned} \underset{\mathbf{v}}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{v} \\ \text{subject to} \quad & |x_i|^p \leq v_i \quad \text{for } i \text{ in } 1:m \end{aligned} \tag{14.18}$$

This form can be optimized using interior point methods (section 10.11).

Example 14.10. A graph implementation for L_2 norm minimization problems.

If our problem is recognized to be a least-squares problem, we can simplify by squaring our objective, thereby removing the square root:

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \quad \Rightarrow \quad \underset{\mathbf{x}}{\text{minimize}} \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

The least-squares problem has a known solution, $\mathbf{x} = \left(\mathbf{A}^\top \mathbf{A}\right)^{-1} \mathbf{A}^\top \mathbf{b}$, and can be solved efficiently as discussed in section 13.2.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

to solve it. Many solver types exist and use techniques already covered in this text.

One common solution technique is to use interior point methods (section 10.11). A DCP can be solved with interior point methods if every atom is either twice differentiable, has a predefined barrier function, or has a graph implementation that is itself solvable with interior point methods.

Interior point methods move all inequality constraints, other than affine constraints, into the objective function using a barrier function:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} + d + \frac{1}{\rho} p_{\text{barrier}}(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \end{aligned} \tag{14.19}$$

The optimal solution approaches the solution of the original problem as ρ approaches infinity. Interior point methods such as algorithm 10.4 iteratively solve this formulation, increasing ρ as they go. Intermediate solutions remain feasible, and are thus in the interior of the feasible set.

Each iteration typically solves equation (14.19) with Newton's method from an initial point $\mathbf{x}$. Here, the search direction $\Delta\mathbf{x}$ is obtained jointly with the dual parameters $\boldsymbol{\mu}$ by solving the *augmented system*:[20]

$$\begin{bmatrix} \nabla^2 p_{\text{barrier}}(\mathbf{x}) & \mathbf{A}^\top \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \boldsymbol{\mu} \end{bmatrix} = - \begin{bmatrix} \rho\mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x}) \\ \mathbf{A}\mathbf{x} - \mathbf{b} \end{bmatrix} = - \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_p \end{bmatrix} \tag{14.20}$$

[20] The augmented system enforces the KKT conditions of stationarity and feasibility. See exercise 14.7.

The left hand side of the augmented system is symmetric but not positive definite. In some cases we know that $\nabla^2 p_{\text{barrier}}(\mathbf{x})$ is positive definite,[21] and we can transform the problem. Here, we rearrange the top row to obtain

[21] The Hessian matrix is positive definite for linear programs in standard form, for example.

$$\Delta\mathbf{x} = -\nabla^2 p_{\text{barrier}}(\mathbf{x})^{-1} \left(\mathbf{r}_d + \mathbf{A}^\top \boldsymbol{\mu} \right) \tag{14.21}$$

which can be substituted into the second row to obtain

$$\left(\mathbf{A} \nabla^2 p_{\text{barrier}}(\mathbf{x})^{-1} \mathbf{A}^\top \right) \boldsymbol{\mu} = \mathbf{r}_p - \mathbf{A} \nabla^2 p_{\text{barrier}}(\mathbf{x})^{-1} r_d \tag{14.22}$$

Inverting an arbitrary $n \times n$ matrix is $O(n^3)$. Unfortunately, canonicalization introduces many additional variables into our problem, resulting in a large Hessian matrix $\nabla^2 p_{\text{barrier}}(\mathbf{x})$. Fortunately, a DCP in partitioned canonical form is composed of many independent atoms that depend on different subsets of our expanded design variables $\mathbf{x}$, resulting in a sparse Hessian matrix with *block-diagonal*

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

structure:

$$\begin{bmatrix} \nabla^2 p_{\text{barrier}}^{(1)}(\mathbf{x}^{(1)}) & & & \\ & \nabla^2 p_{\text{barrier}}^{(2)}(\mathbf{x}^{(2)}) & & \\ & & \ddots & \\ & & & \nabla^2 p_{\text{barrier}}^{(k)}(\mathbf{x}^{(k)}) \end{bmatrix} \tag{14.23}$$

The inverse of a block-diagonal matrix is simply a block-diagonal matrix whose entries are the inverses of each block. By exploiting this structure, the inversion becomes extremely efficient.

When we cannot invert our Hessian, we solve the augmented system with an $\mathbf{L}\mathbf{D}\mathbf{L}^\top$ factorization[22] with a lower triangular $\mathbf{L}$ and a block-diagonal $\mathbf{D}$. This factorization can often be computed very efficiently, again leveraging block-diagonal structure. Additional tricks, including properties of selected barrier functions, can also be applied.

[22] This factorization is covered in appendix C.8.2.

14.5 Summary

- A disciplined convex program is a type of convex optimization problem for which automated methods can verify convexity and transcribe it into a canonical form amenable to solution by existing solvers.
- Convexity for a DCP requires that its expressions be product-free and that they follow sign, composition, and top-level production rules.
- Canonicalization is the process of taking a DCP and producing a problem in partitioned canonical form.
- Interior-point methods are popular for solving problems in partitioned canonical form because they can efficiently exploit the underlying block-diagonal structure.

14.6 Exercises

Exercise 14.1. We can use extended-valued functions to conveniently extend functions over a subset of $\mathbb{R}^n$ to all of $\mathbb{R}^n$. This extension works by having the function produce an infinite value if it receives an input outside of its typical domain, thereby extending its range to include infinite values.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Show that if we have a convex function f defined over a convex set $\mathcal{S}$, then its extended-value extension

$$g(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{if } x \in \mathcal{S} \\ \infty & \text{otherwise} \end{cases}$$

is also convex.

Solution: According to the definition given in appendix C.3, our extended-valued function g is convex over $\mathbb{R}^n$ if, for all $\mathbf{x}, \mathbf{y}$ in $\mathbb{R}^n$ and for all α in $[0, 1]$,

$$g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha g(\mathbf{x}) + (1-\alpha)g(\mathbf{y})$$

If both $\mathbf{x}$ and $\mathbf{y}$ are in $\mathcal{S}$, then $\alpha\mathbf{x} + (1-\alpha)\mathbf{y} \in \mathcal{S}$ because $\mathcal{S}$ is convex. Furthermore, we know

$$f(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha)f(\mathbf{y})$$

is satisfied because f is convex.

If $\mathbf{x} \notin \mathcal{S}$ and $\mathbf{y} \in \mathcal{S}$, then we get:

$$g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha \cdot \infty + (1-\alpha)f(\mathbf{y})$$

which is $g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \infty$ for $\alpha > 0$ and $f(\mathbf{y}) = f(\mathbf{y})$ when $\alpha = 0$, so it holds.

Similarly, if $\mathbf{x} \in \mathcal{S}$ and $\mathbf{y} \notin \mathcal{S}$, then we get:

$$g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1-\alpha) \cdot \infty$$

which is $g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \infty$ for $\alpha < 1$ and $f(\mathbf{x}) = f(\mathbf{x})$ when $\alpha = 1$, so it holds.

Finally, if both $\mathbf{x}$ and $\mathbf{y}$ are not in $\mathcal{S}$, then we get:

$$g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \alpha \cdot \infty + (1-\alpha) \cdot \infty$$

which is $g(\alpha\mathbf{x} + (1-\alpha)\mathbf{y}) \leq \infty$ for $0 \leq \alpha \leq 1$, so it holds.

Exercise 14.2. The product-free expressions do not allow for products, so $x \cdot x$ or x^2 is not DCP-compliant. Show how squares could still be incorporated into disciplined convex programming.

Solution: We can introduce a new atom to support squares. Let us call this atom `square`(x), and use it to replace all instances of $x \cdot x$ or x^2. The atom is convex, increasing in x for $x \geq 0$, decreasing in x for $x \leq 0$, and has range ≥ 0. It will then be treated just like other atoms, and we can successfully solve problems containing squares.

Exercise 14.3. Linearize the optimization problem from example 14.6:

$$\begin{aligned} \underset{x_1, x_2}{\text{minimize}} \quad & \text{pow}(a_1 x_1 + a_2 x_2 - b_1, 2) \\ \text{subject to} \quad & c_1 x_1 + c_2 x_2 = d \\ & \sqrt{\min(x_1, x_2)} \geq 1 - \log(x_2) \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: First we linearize the objective function, introducing v_1 and $u_{1,1}$:

$$\text{pow}(a_1x_1 + a_2x_2 - b_1, 2) \quad \Rightarrow \quad \begin{aligned} \underset{x_1,x_2,v_1,u_{1,1}}{\text{minimize}} \quad & v_1 \\ \text{subject to} \quad & u_{1,1} = a_1x_1 + a_2x_2 - b_1 \\ & \text{pow}(u_{1,1}, 2) \leq v_1 \end{aligned}$$

The affine constraint does not need to be linearized, so we next linearize the left-hand side of the inequality constraint. We start by recursing in and linearizing $\min(x_1, x_2)$, introducing v_2, $u_{2,1}$, and $u_{2,2}$:

$$\min(x_1, x_2) \quad \Rightarrow \quad \begin{aligned} \text{minimize} \quad & v_2 \\ \text{subject to} \quad & u_{2,1} = x_1 \\ & u_{2,2} = x_2 \\ & \min(u_{2,1}, u_{2,2}) \geq v_2 \end{aligned}$$

We then linearize $\sqrt{v_2}$, introducing v_3 and $u_{3,1}$:

$$\sqrt{v_2} \quad \Rightarrow \quad \begin{aligned} \text{minimize} \quad & v_3 \\ \text{subject to} \quad & u_{3,1} = v_2 \\ & \sqrt{u_{3,1}} \geq v_3 \end{aligned}$$

We next linearize the right-hand side of the inequality constraint, introducing v_4 and $u_{4,1}$:

$$1 - \log(x_2) \quad \Rightarrow \quad \begin{aligned} \text{minimize} \quad & 1 - v_4 \\ \text{subject to} \quad & u_{4,1} = x_2 \\ & \log(u_{4,1}) \geq v_4 \end{aligned}$$

We conclude by introducing a slack variable $s \geq 0$ to make our top-level inequality constraint linear:

$$v_3 \geq 1 - v_4 \quad \Rightarrow \quad v_3 = 1 - v_4 + s$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Our linearized problem is thus:

$$\begin{aligned}
\underset{\mathbf{x}_{1:2}, u_{1,1}, \mathbf{u}_{2,1:2}, u_{3,1}, u_{4,1}, \mathbf{v}_{1:4}, s}{\text{minimize}} \quad & v_1 \\
\text{subject to} \quad & c_1 x_1 + c_2 x_2 = d \\
& v_3 = 1 - v_4 + s \\
& \sqrt{u_{3,1}} \geq v_3 \\
& \text{pow}(u_{1,1}, 2) \leq v_1 \\
& \min(u_{2,1}, u_{2,2}) \geq v_2 \\
& \log(u_{4,1}) \geq v_4 \\
& s \geq 0 \\
& u_{1,1} = a_1 x_1 + a_2 x_2 - b_1 \\
& u_{2,1} = x_1 \\
& u_{2,2} = x_2 \\
& u_{3,1} = v_2 \\
& u_{4,1} = x_2
\end{aligned}$$

Exercise 14.4. Convert the graph-expanded p-norm problem in example 14.9 into an unconstrained problem using log barriers:[23]

[23] Log barrier functions are given in equation (10.40).

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{v}}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{v} \\
\text{subject to} \quad & |\mathbf{a}_i^\top \mathbf{x} - b_i|^p \leq v_i \qquad \text{for } i \text{ in } 1:m
\end{aligned}$$

In doing so, please remove the absolute value.

Solution: We first exponentiate the inequalities by $2/p$ in order to drop the absolute value:[24]

[24] We know that both sides of the inequality are nonnegative. We want to exponentiate with a positive value to avoid changing the sign. Since $p \geq 1$, we know that $2/p \geq 0$.

$$|\mathbf{a}_i^\top \mathbf{x} - b_i|^p \leq v_i \quad \Rightarrow \quad |\mathbf{a}_i^\top \mathbf{x} - b_i|^2 \leq v_i^{2/p}$$

We then move the right-hand side to the left to obtain the form $g(\cdot) \leq 0$:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{v}}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{v} \\
\text{subject to} \quad & \left(\mathbf{a}_i^\top \mathbf{x} - b_i\right)^2 - v_i^{2/p} \leq 0 \qquad \text{for } i \text{ in } 1:m
\end{aligned}$$

We can then construct a log barrier for the constraints using equation (10.40):[25]

[25] Note that we use ∞ to produce an extended real-valued function.

$$p_{\text{barrier}}(\mathbf{x}, \mathbf{v}) = -\sum_i \begin{cases} \log\left(v_i^{2/p} - \left(\mathbf{a}_i^\top \mathbf{x} - b_i\right)^2\right) & \text{if } (\mathbf{x}, \mathbf{v}) \text{ is feasible with respect to constraint } i \\ \infty & \text{otherwise} \end{cases}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 14.5. Consider the atom $f(x) = \sqrt{x}$ for $x \geq 0$. Derive a graph implementation that avoids the nondifferentiability at $x = 0$.

Solution: Because we know $\sqrt{x}$ is concave, we use its hypograph:

$$\text{hypo}\sqrt{x} = \{(x, y) \mid x \geq 0, y \leq \sqrt{x}\}$$

We can equivalently express the hypograph as:

$$\text{hypo}\sqrt{x} = \left\{(x, y) \mid x \geq 0, \max(y, 0)^2 \leq x\right\}$$

We can drop $x \geq 0$ because it will automatically be satisfied when $\max(y, 0)^2 \leq x$ is satisfied. We can drop the max because y^2 is already nonnegative. These changes result in the following graph implementation:

$$\sqrt{x} \quad \Rightarrow \quad \begin{array}{ll} \underset{y}{\arg\max} & y \\ \text{subject to} & y^2 \leq x \end{array}$$

The graph implementation is differentiable everywhere.

Exercise 14.6. Canonicalize the following convex problem and apply the graph implementations from example 14.9:

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 \\ \text{subject to} & \|\mathbf{C}\mathbf{x} - \mathbf{d}\|_{1.5} \leq 3 \end{array}$$

where

$$\mathbf{A} = \begin{bmatrix} 2 & -1 \\ 1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 4 \\ -5 \end{bmatrix}$$
$$\mathbf{C} = \begin{bmatrix} 0 & 1 \\ 3 & -4 \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Solution: The norms are both convex, and because they contain affine expressions, the overall expressions are convex.

Linearizing the problem produces:

$$\begin{array}{lr} \underset{\mathbf{x},\mathbf{v},s}{\text{minimize}} & v_1 \\ \text{subject to} & v_1 = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 \\ & v_2 = \|\mathbf{C}\mathbf{x} - \mathbf{d}\|_{1.5} \\ & v_2 + s = 3 \\ & s \geq 0 \end{array}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We then apply a graph expansion for the L_1 norm:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{y}^{(1)},\mathbf{v},s}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{y}^{(1)} \\
\text{subject to} \quad & -\mathbf{y}^{(1)} \leq \mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{y}^{(1)} \\
& v_2 = \|\mathbf{C}\mathbf{x} - \mathbf{d}\|_{1.5} \\
& v_2 + s = 3 \\
& s \geq 0
\end{aligned}$$

We then apply a graph expansion for the L_p norm:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{y}^{(1)},\mathbf{y}^{(2)},\mathbf{v},s}{\text{minimize}} \quad & \mathbf{1}^\top \mathbf{y}^{(1)} + \mathbf{1}^\top \mathbf{y}^{(2)} \\
\text{subject to} \quad & -\mathbf{y}^{(1)} \leq \mathbf{A}\mathbf{x} - \mathbf{b} \leq \mathbf{y}^{(1)} \\
& |x_2 + 1|^{1.5} \leq y_1^{(2)} \\
& |3x_1 - 4x_2|^{1.5} \leq y_2^{(2)} \\
& \mathbf{1}^\top \mathbf{y}^{(2)} + s = 3 \\
& s \geq 0
\end{aligned}$$

Exercise 14.7. The augmented system in equation (14.20) comes from applying Newton's method to equation (14.19). Derive the augmented system. Start by multiplying the objective function by ρ and then take the second-order Taylor approximation to obtain an objective function in terms of an offset $\Delta\mathbf{x}$ from the current design. The augmented system can then be obtained from the KKT conditions; the first row comes from stationarity and the second from the original constraint.

Solution: We start by multiplying the objective function by ρ:

$$\begin{aligned}
\underset{\mathbf{x}}{\text{minimize}} \quad & \rho\mathbf{c}^\top \mathbf{x} + \rho d + p_{\text{barrier}}(\mathbf{x}) \\
\text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b}
\end{aligned}$$

To apply Newton's method, we replace the original objective f with its second-order Taylor approximation about the current design $\mathbf{x}$:

$$\begin{aligned}
\hat{f}(\Delta\mathbf{x}) &= f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \nabla^2 f(\mathbf{x})\Delta\mathbf{x} \\
&= \rho\mathbf{c}^\top \mathbf{x} + \rho d + p_{\text{barrier}}(\mathbf{x}) + (\rho\mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x}))^\top \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \nabla^2 p_{\text{barrier}}(\mathbf{x})\Delta\mathbf{x}
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

From the KKT conditions, we know that a solution must satisfy both $\nabla \hat{f}(\Delta \mathbf{x}) + \mathbf{A}^\top \boldsymbol{\mu} = \mathbf{0}$ and $\mathbf{A}(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{b}$. The first constraint works out to be:

$$\begin{aligned} \nabla \hat{f}(\Delta \mathbf{x}) + \mathbf{A}^\top \boldsymbol{\mu} &= \mathbf{0} \\ \rho \mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x}) + \nabla^2 p_{\text{barrier}}(\mathbf{x}) \Delta \mathbf{x} + \mathbf{A}^\top \boldsymbol{\mu} &= \mathbf{0} \\ \nabla^2 p_{\text{barrier}}(\mathbf{x}) \Delta \mathbf{x} + \mathbf{A}^\top \boldsymbol{\mu} &= -(\rho \mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x})) \end{aligned}$$

If we include the second constraint we get:

$$\begin{aligned} \nabla^2 p_{\text{barrier}}(\mathbf{x}) \Delta \mathbf{x} + \mathbf{A}^\top \boldsymbol{\mu} &= -(\rho \mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x})) \\ \mathbf{A}(\mathbf{x} + \Delta \mathbf{x}) &= \mathbf{b} \end{aligned}$$

which produces the augmented system:

$$\begin{bmatrix} \nabla^2 p_{\text{barrier}}(\mathbf{x}) & \mathbf{A}^\top \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \boldsymbol{\mu} \end{bmatrix} = - \begin{bmatrix} \rho \mathbf{c} + \nabla p_{\text{barrier}}(\mathbf{x}) \\ \mathbf{A}\mathbf{x} - \mathbf{b} \end{bmatrix}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

15 Multiobjective Optimization

Previous chapters have developed methods for optimizing single-objective functions, but this chapter is concerned with *multiobjective optimization*, or *vector optimization*, where we must optimize with respect to several objectives simultaneously. Engineering is often a trade-off between cost, performance, and time-to-market, and it is often unclear how to prioritize different objectives. We will discuss various methods for transforming vector-valued objective functions to scalar-valued objective functions so that we can use the algorithms discussed in previous chapters to arrive at an optimum. In addition, we will discuss algorithms for identifying the set of design points that represent the best trade-off between objectives, without having to commit to a particular prioritization of objectives. These design points can then be presented to experts who can then identify the most desirable design.[1]

[1] Additional methods are surveyed in R. T. Marler and J. S. Arora, "Survey of Multi-Objective Optimization Methods for Engineering," *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, 2004. For a textbook dedicated entirely to multiobjective optimization, see K. Miettinen, *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, 1999.

15.1 Pareto Optimality

The notion of *Pareto optimality* is useful when discussing problems where there are multiple objectives. A design is Pareto optimal if it is impossible to improve in one objective without worsening at least one other objective. In multiobjective design optimization, we can generally focus our efforts on designs that are Pareto optimal without having to commit to a particular trade-off between objectives. This section introduces some definitions and concepts that are helpful when discussing approaches to identifying Pareto-optimal designs.

15.1.1 Dominance

In single-objective optimization, two design points $\mathbf{x}$ and $\mathbf{x}'$ can be ranked objectively based on their scalar function values. The point $\mathbf{x}'$ is better whenever $f(\mathbf{x}')$ is less than $f(\mathbf{x})$.

In multiobjective optimization, our objective function $\mathbf{f}$ returns an m-dimensional vector of values $\mathbf{y}$ when evaluated at a design point $\mathbf{x}$. The different dimensions of $\mathbf{y}$ correspond to different objectives, sometimes also referred to as metrics or criteria. We can objectively rank two design points $\mathbf{x}$ and $\mathbf{x}'$ only when one is better in at least one objective and no worse in any other. That is, $\mathbf{x}$ *dominates* $\mathbf{x}'$ if and only if

$$\begin{aligned} f_i(\mathbf{x}) &\leq f_i(\mathbf{x}') \text{ for } i \text{ in } 1:m \\ \text{and } f_i(\mathbf{x}) &< f_i(\mathbf{x}') \text{ for some } i \end{aligned} \tag{15.1}$$

as compactly implemented in algorithm 15.1.

```
dominates(y, y′) = all(y .≤ y′) && any(y .< y′)
```

Algorithm 15.1. A method for checking whether $\mathbf{x}$ dominates $\mathbf{x}'$, where `y` is the vector of objective values for $\mathbf{f}(\mathbf{x})$ and `y′` is the vector of objective values for $\mathbf{f}(\mathbf{x}')$.

Figure 15.1 shows that in multiple dimensions there are regions with dominance ambiguity. This ambiguity arises whenever $\mathbf{x}$ is better in some objectives and $\mathbf{x}'$ is better in others. Several methods exist for resolving these ambiguities.

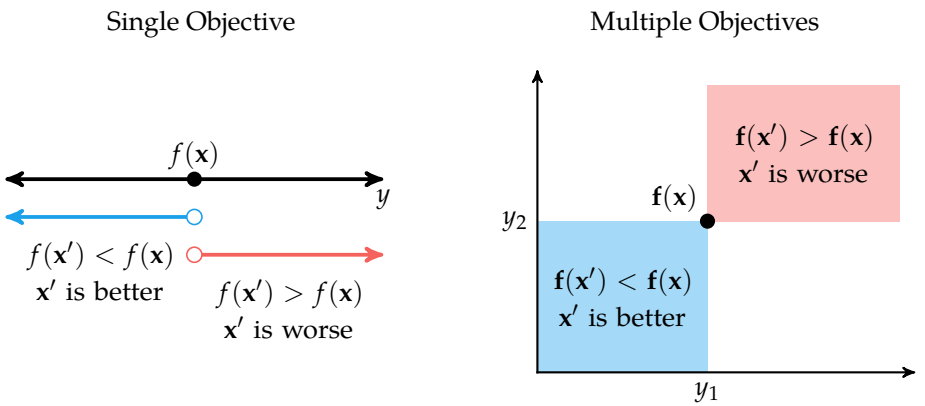


Figure 15.1. Design points can be objectively ranked in single-objective optimization but can be objectively ranked in multi-objective optimization only in some cases.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

15.1.2 Pareto Frontier

In mathematics, an *image* of an input set through some function is the set of all possible outputs of that function when evaluated on elements of that input set. We will denote the image of $\mathcal{X}$ through $\mathbf{f}$ as $\mathcal{Y}$, and we will refer to $\mathcal{Y}$ as the *criterion space*. Section 15.1.2 shows examples of criterion space for problems with single and multiple objectives. As illustrated, the criterion space in single-objective optimization is one dimensional. All of the global optima share a single objective function value, y^*. In multiobjective optimization, the criterion space is m-dimensional, where m is the number of objectives. There is typically no globally best objective function value because there may be ambiguity when trade-offs between objectives are not specified.

In multiobjective optimization, we can define the notion of *Pareto optimality*. A design point $\mathbf{x}$ is Pareto-optimal when no point dominates it. That is, $\mathbf{x} \in \mathcal{X}$ is Pareto-optimal if there does not exist an $\mathbf{x}' \in \mathcal{X}$ such that $\mathbf{x}'$ dominates $\mathbf{x}$. The set of Pareto-optimal points forms the *Pareto frontier*,[2] and all these points lie on the boundary of the criterion space. The Pareto frontier is valuable in making design trade decisions as discussed in example 15.1.

Some multiobjective optimization methods also find *weakly Pareto-optimal* points. Whereas Pareto-optimal points are those such that no other point improves at least one objective without worsening another, weakly Pareto-optimal points are those such that no other point improves all of the objectives (figure 15.3). That is, $\mathbf{x} \in \mathcal{X}$ is weakly Pareto-optimal if there does not exist an $\mathbf{x}' \in \mathcal{X}$ such that $\mathbf{f}(\mathbf{x}') < \mathbf{f}(\mathbf{x})$. Pareto-optimal points are also weakly Pareto optimal. Weakly Pareto-optimal points are not necessarily Pareto optimal.

We define the *utopia point* to be the point in the objective space consisting of the component-wise optima:

$$y_i^{\text{utopia}} = \underset{\mathbf{x} \in \mathcal{X}}{\text{minimize}}\, f_i(\mathbf{x}) \tag{15.2}$$

The utopia point is often not attainable; optimizing one component typically requires a trade-off in another component.

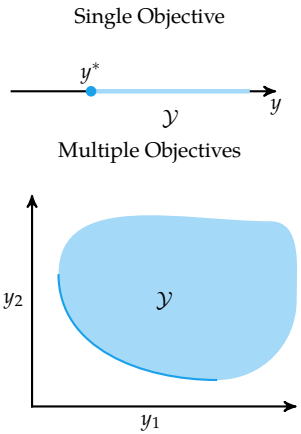


Figure 15.2. The criterion space is the set of all objective values obtained by feasible design points. The Pareto frontier is highlighted in dark blue.

[2] In two dimensions, the Pareto frontier is also referred to as a *Pareto curve*.

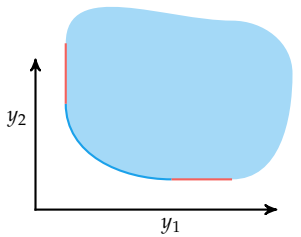


Figure 15.3. Weakly Pareto-optimal points, shown in red, cannot be improved simultaneously in all objectives.

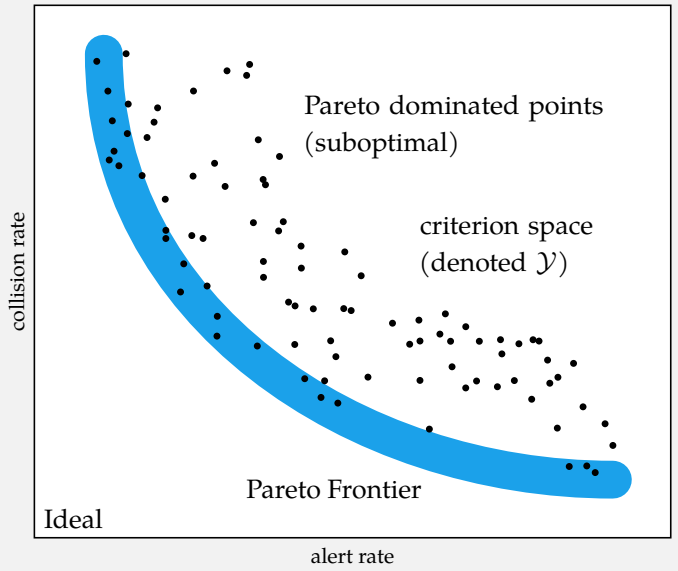
15.1.3 Pareto Frontier Generation

There are several methods for generating Pareto frontiers. A naive approach is to sample design points throughout the design space and then to identify

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 15.1. An approximate Pareto frontier obtained from evaluating many different design points for an aircraft collision avoidance system.

When constructing a collision avoidance system for aircraft, we must minimize both the collision rate and the alert rate. Although more alerts can result in preventing more collisions if the alerts are followed, too many alerts can result in pilots losing trust in the system and decrease compliance. Hence, the designers of the system must carefully trade alerts and collision rate.



By varying the design parameters, we can obtain many different collision avoidance systems, but some of these will be better than others. A Pareto frontier can be extracted to help understand the effects that objective trade-offs will have on the system.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

the nondominated points (algorithm 15.2). This approach is typically wasteful, leading to many dominated design points as shown in figure 15.4. In addition, this approach does not guarantee a smooth or correct Pareto frontier. The remainder of this chapter discusses more effective ways to generate Pareto frontiers.

```
function naive_pareto(xs, ys)
    pareto_xs, pareto_ys = similar(xs, 0), similar(ys, 0)
    for (x,y) in zip(xs,ys)
        if !any(dominates(y′,y) for y′ in ys)
            push!(pareto_xs, x)
            push!(pareto_ys, y)
        end
    end
    return (pareto_xs, pareto_ys)
end
```

Algorithm 15.2. A method for generating a Pareto frontier using randomly sampled design points `xs` and their multiobjective values `ys`. Both the Pareto-optimal design points and their objective values are returned.

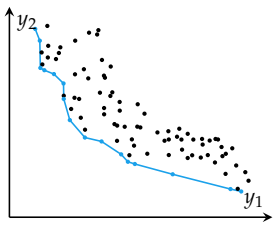


Figure 15.4. Generating Pareto frontiers with naively scattered points is straightforward but inefficient and approximate.

15.2 Constraint Methods

Constraints can be used to cut out sections of the Pareto frontier and obtain a single point in the criterion space. Constraints can be supplied either by the designer or automatically obtained based on an ordering of the objectives.

15.2.1 Constraint Method

The *constraint method* constrains all but one of the objectives. Here we choose f_1 without loss of generality:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f_1(\mathbf{x}) \\ \text{subject to} \quad & f_2(\mathbf{x}) \leq c_2 \\ & \vdots \\ & f_m(\mathbf{x}) \leq c_m \\ & \mathbf{x} \in \mathcal{X} \end{aligned} \tag{15.3}$$

Given the vector $\mathbf{c}$, the constraint method produces a unique optimal point in the criterion space, provided that the constraints are feasible. The constraint method can be used to generate Pareto frontiers by varying $\mathbf{c}$ as shown in figure 15.5.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

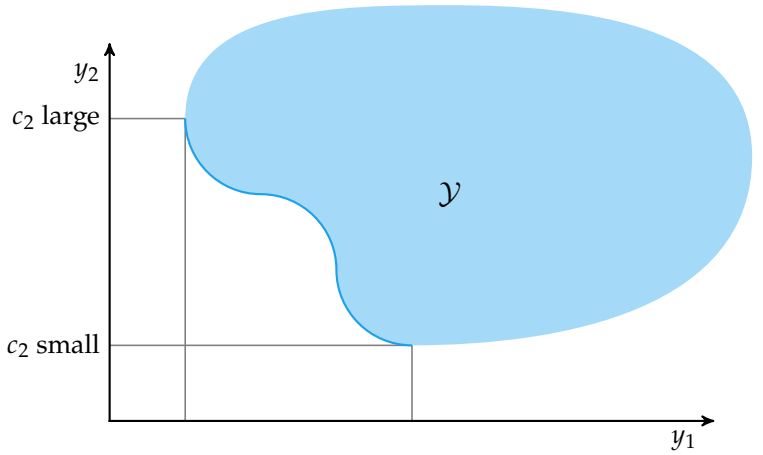


Figure 15.5. The constraint method for generating a Pareto frontier. This method can identify points in the concave region of the Pareto frontier.

15.2.2 *Lexicographic Method*

The *lexicographic method* ranks the objectives in order of importance. A series of single-objective optimizations are performed on the objectives in order of importance. Each optimization problem includes constraints to preserve the optimality with respect to previously optimized objectives as shown in figure 15.6.

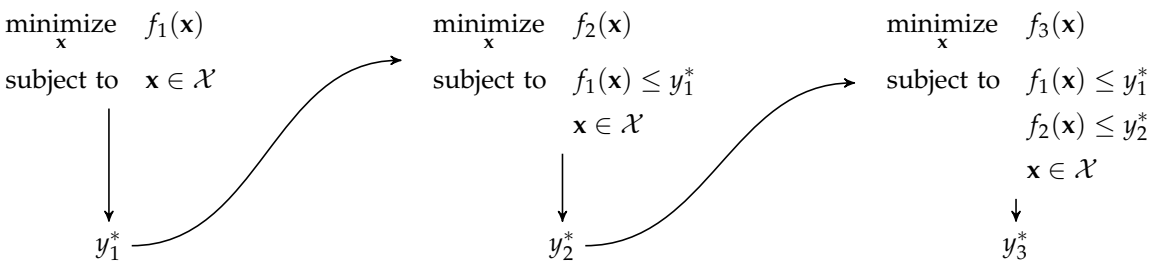


Figure 15.6. The lexicographic method for an optimization problem with three objectives.

Iterations are always feasible because the minimum point from the previous optimization is always feasible. The constraints could also be replaced with equalities, but inequalities are often easier for optimizers to enforce. In addition, if the optimization method used is not optimal, then subsequent optimizations may encounter better solutions that would otherwise be rejected. The lexicographic method is sensitive to the ordering of the objective functions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

15.3 Weight Methods

A designer can sometimes identify preferences between the objectives and encode these preferences as a vector of weights. In cases where the choice of weights is not obvious, we can generate a Pareto frontier by sweeping over the space of weights and solving the resulting weighted optimization problems. This section also discusses a variety of alternative methods for transforming multiobjective functions into single-objective functions.

15.3.1 Weighted Sum Method

The *weighted sum method* (algorithm 15.3) uses a vector of weights $\mathbf{w}$ to convert $\mathbf{f}$ to a single objective f:[3]

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{f}(\mathbf{x}) \tag{15.4}$$

where the weights are nonnegative and sum to 1. The weights can be interpreted as costs associated with each objective. The Pareto frontier can be extracted by varying $\mathbf{w}$ and solving the associated optimization problem with the objective in equation (15.4). In two dimensions, we vary w_1 from 0 to 1, setting $w_2 = 1 - w_1$. This approach is illustrated in figure 15.7.

[3] L. Zadeh, "Optimality and Non-Scalar-Valued Performance Criteria," *IEEE Transactions on Automatic Control*, vol. 8, no. 1, pp. 59–60, 1963.

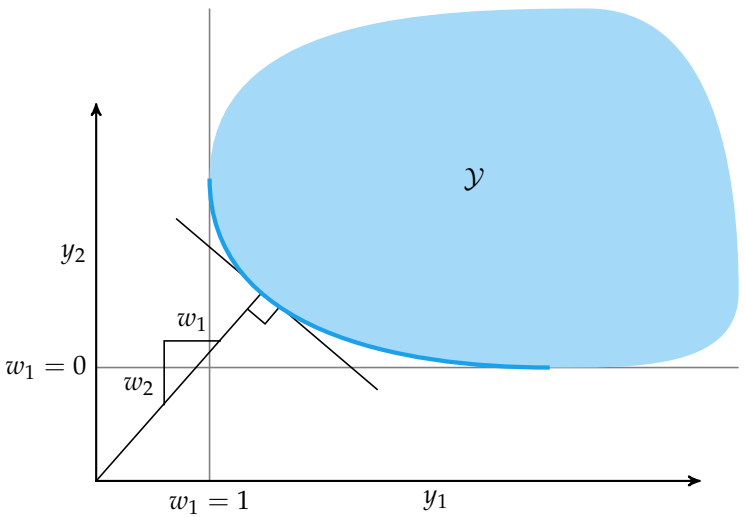


Figure 15.7. The weighted sum method used to generate a Pareto frontier. Varying the weights allows us to trace the Pareto frontier.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In contrast with the constraint method, the weighted sum method cannot obtain points in nonconvex regions of the Pareto frontier as shown in figure 15.8. For some problems, it may be desirable to obtain points in these regions. A given set of weights forms a linear objective function with parallel contour lines marching away from the origin. If the feasible set bends away from the origin, it will have other Pareto optimal points on the boundary that cannot be recovered by minimizing equation (15.4).

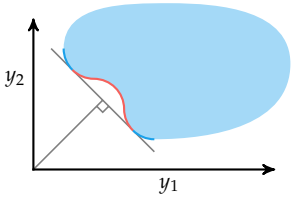


Figure 15.8. The points in red are Pareto optimal but cannot be obtained using the weighted sum method.

```
function weight_pareto(f1, f2, npts)
    return [
        optimize(x->w1*f1(x) + (1-w1)*f2(x))
        for w1 in range(0,stop=1,length=npts)
    ]
end
```

Algorithm 15.3. The weighted sum method for generating a Pareto frontier, which takes objective functions `f1` and `f2` and number of Pareto points `npts`.

15.3.2 Goal Programming

Goal programming[4] is a method for converting a multiobjective function to a single-objective function by minimizing an L_p norm[5] between $\mathbf{f}(\mathbf{x})$ and a goal point:

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}} \left\| \mathbf{f}(\mathbf{x}) - \mathbf{y}^{\text{goal}} \right\|_p \tag{15.5}$$

where the goal point is typically the utopia point. The equation above does not involve a vector of weights, but the other methods discussed in this chapter can be thought of as generalizations of goal programming. This approach is illustrated in figure 15.9.

[4] An overview is presented in D. Jones and M. Tamiz, *Practical Goal Programming*. Springer, 2010.

[5] The definition of L_p-norms is covered in appendix C.4. Goal programming typically uses $p = 1$.

15.3.3 Weighted Exponential Sum

The *weighted exponential sum* combines goal programming and the weighted sum method[6]

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i \left(f_i(\mathbf{x}) - y_i^{\text{goal}} \right)^p \tag{15.6}$$

where $\mathbf{w}$ is a vector of positive weights that sum to 1 and $p \geq 1$ is an exponent similar to that used in L_p norms. Zero-valued weights can result in weakly Pareto-optimal points.

[6] P. L. Yu, "Cone Convexity, Cone Extreme Points, and Nondominated Solutions in Decision Problems with Multiobjectives," *Journal of Optimization Theory and Applications*, vol. 14, no. 3, pp. 319–377, 1974.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

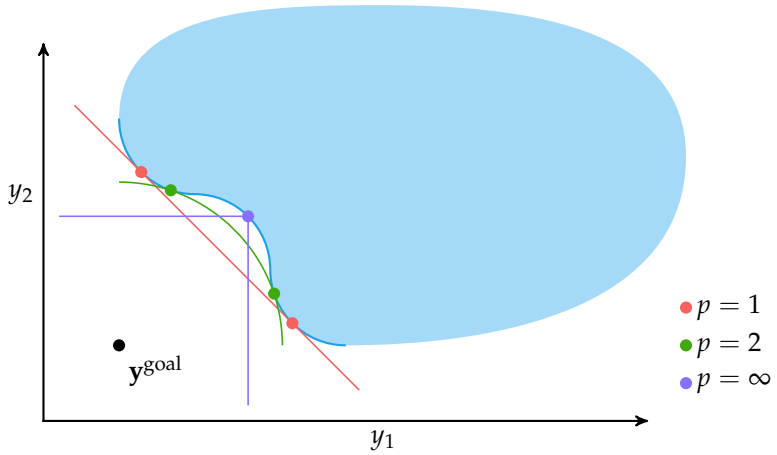


Figure 15.9. Solutions to goal programming as the value for p is changed.

The weighted exponential sum weighs each component of the distance between the solution point and the goal point in the criterion space. Increasing p increases the relative penalty of the largest coordinate deviation between $\mathbf{f}(\mathbf{x})$ and the goal point. While portions of the Pareto-optimal set can be obtained by continuously varying p, we are not guaranteed to obtain the complete Pareto frontier, and it is generally preferable to vary $\mathbf{w}$ using a constant p.

15.3.4 *Weighted Min-Max Method*

Using higher values of p with the weighted exponential sum objective tends to produce better coverage of the Pareto frontier because the distance contours are able to enter nonconvex regions of the Pareto frontier. The *weighted min-max method*, also called the *weighted Tchebycheff method*, is the limit as p approaches infinity:[7]

$$f(\mathbf{x}) = \max_i \left[w_i \left(f_i(\mathbf{x}) - y_i^{\text{goal}} \right) \right] \tag{15.7}$$

The weighted min-max method can provide the complete Pareto-optimal set by scanning over the weights but will also produce weakly Pareto-optimal points. The method can be augmented to produce only the Pareto frontier

$$f(\mathbf{x}) = \max_i \left[w_i \left(f_i(\mathbf{x}) - y_i^{\text{goal}} \right) \right] + \rho \, \mathbf{f}(\mathbf{x})^\top \mathbf{y}^{\text{goal}} \tag{15.8}$$

[7] The maximization can be removed by including an additional parameter λ:

$$\begin{aligned} \underset{\mathbf{x},\lambda}{\text{minimize}} \quad & \lambda \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \\ & \mathbf{w} \odot \left(\mathbf{f}(\mathbf{x}) - \mathbf{y}^{\text{goal}} \right) - \lambda \mathbf{1} \leq \mathbf{0} \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where ρ is a small positive scalar with values typically between 0.0001 and 0.01. The added term requires that all terms in $\mathbf{y}^{\text{goal}}$ be positive, which can be accomplished by shifting the objective function. By definition, $\mathbf{f}(\mathbf{x}) \geq \mathbf{y}^{\text{goal}}$ for all $\mathbf{x}$. Any weakly Pareto-optimal point will have $\mathbf{f}(\mathbf{x})^\top \mathbf{y}^{\text{goal}}$ larger than a strongly Pareto-optimal point closer to $\mathbf{y}^{\text{goal}}$.

15.3.5 Exponential Weighted Criterion

The *exponential weighted criterion*[8] was motivated by the inability of the weighted sum method to obtain points on nonconvex portions of the Pareto frontier. It constructs a scalar objective function according to

$$f(\mathbf{x}) = \sum_{i=1}^{m} (e^{pw_i} - 1) e^{pf_i(\mathbf{x})} \tag{15.9}$$

Each objective is individually transformed and reweighted. High values of p can lead to numerical overflow.

[8] T. W. Athan and P. Y. Papalambros, "A Note on Weighted Criteria Methods for Compromise Solutions in Multi-Objective Optimization," *Engineering Optimization*, vol. 27, no. 2, pp. 155–176, 1996.

15.4 Multiobjective Population Methods

Population methods have also been applied to multiobjective optimization.[9] We can adapt the standard algorithms to encourage populations to spread over the Pareto frontier.

[9] Population methods are covered in chapter 9.

15.4.1 Subpopulations

Population methods can divide their attention over several potentially competing objectives. The population can be partitioned into *subpopulations*, where each subpopulation is optimized with respect to different objectives. A traditional genetic algorithm, for example, can be modified to bias the selection of individuals for recombination toward the fittest individuals within each subpopulation. Those selected can form offspring with individuals from different subpopulations.

One of the first adaptations of population methods to multiobjective optimization is the *vector evaluated genetic algorithm*[10] (algorithm 15.4). Figure 15.10 shows how subpopulations are used in a vector evaluated genetic algorithm to maintain diversity over multiple objectives. The progression of a vector evaluated genetic algorithm is shown in figure 15.11.

[10] J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," in *International Conference on Genetic Algorithms and Their Applications*, 1985.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

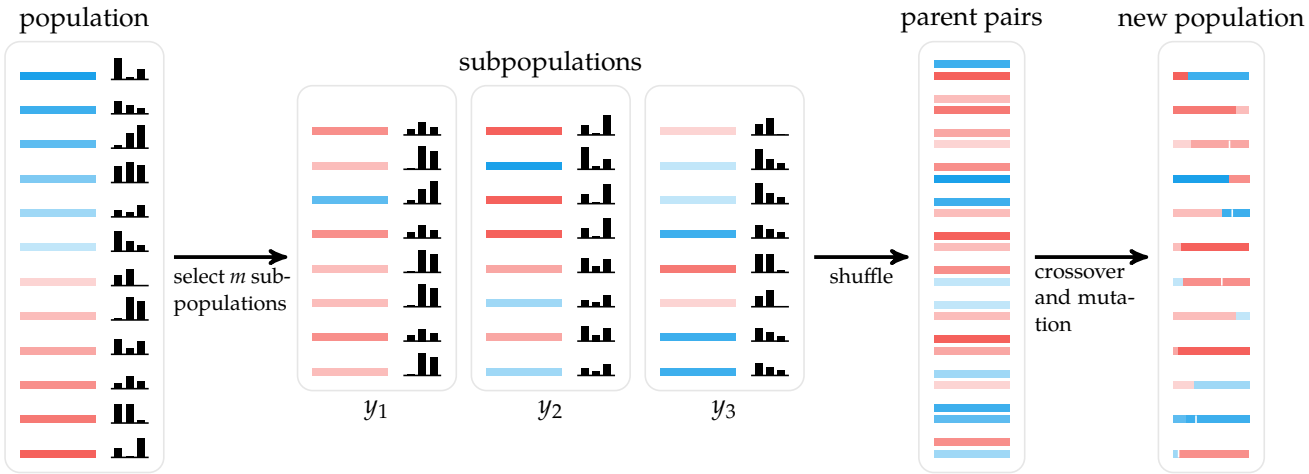


Figure 15.10. Using subpopulations in a vector evaluated genetic algorithm with a three-dimensional objective. Note that smaller objective function values, represented by smaller vertical bars, are better.

```
function vector_evaluated_genetic_algorithm(f, population,
    k_max, S, C, M)

    m = length(f(population[1]))
    m_pop = length(population)
    m_subpop = m_pop ÷ m
    for k in 1:k_max
        ys = f.(population)

        # Select parents for each subpopulation and
        # flatten all parent pairs into one vector of length 2m_pop
        parents = vcat((
            vcat(select(S, [y[i] for y in ys])[1:m_subpop]...)
            for i in 1:m
        )...)

        p = randperm(2m_pop)
        population = [
            mutate(M, crossover(C,
               population[parents[p[2i-1]]],
               population[parents[p[2i]]]))
            for i in 1:m_pop]
    end
    return population
end
```

Algorithm 15.4. The vector evaluated genetic algorithm, which takes a vector-valued objective function `f`, an initial population, a number of iterations `k_max`, a `SelectionMethod S`, a `CrossoverMethod C`, and a `MutationMethod M`. The algorithm selects subpopulations for the m different objective dimensions, then constructs the next population by randomly combining all parents. The resulting population is returned.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

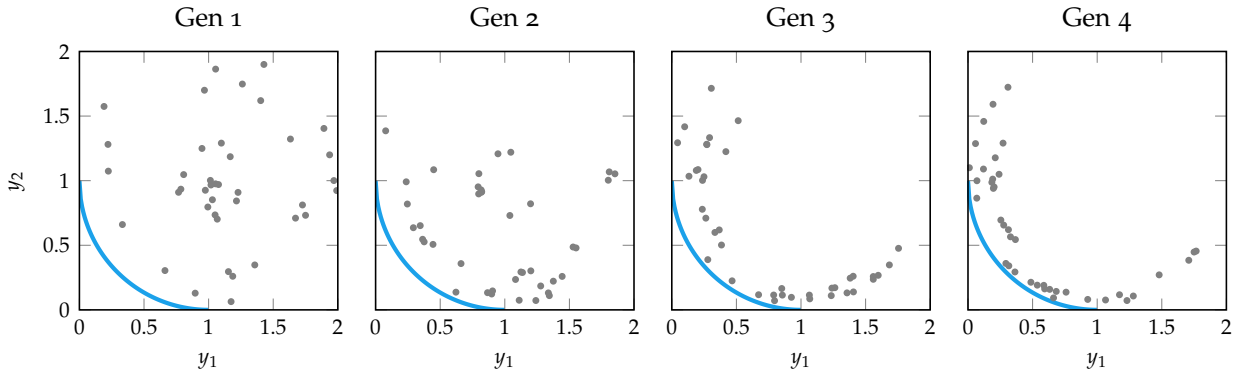


Figure 15.11. A vector evaluated genetic algorithm applied to the circle function defined in appendix B.8. The Pareto frontier is shown in blue.

15.4.2 *Nondomination Ranking*

We can compute naive Pareto frontiers using the individuals in a population. A design point that lies on the approximate Pareto frontier is considered better than a value deep within the criterion space. We can use *nondomination ranking* (algorithm 15.5) to rank individuals according to the following levels:[11]

Level 1. Nondominated individuals in the population.

Level 2. Nondominated individuals except those in Level 1.

Level 3. Nondominated individuals except those in Levels 1 or 2.

⋮

Level k. Nondominated individuals except those in Levels 1 to $k-1$.

Level 1 is obtained by applying algorithm 15.2 to the population. Subsequent levels are generated by removing all previous levels from the population and then applying algorithm 15.2 again. This process is repeated until all individuals have been ranked. An individual's objective function value is proportional to its rank. The nondomination levels for an example population are shown in figure 15.12.

[11] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

15.4.3 *Pareto Filters*

Population methods can be augmented with a *Pareto filter*, which is a population that approximates the Pareto frontier.[12] The filter is typically updated with every

[12] H. Ishibuchi and T. Murata, "A Multi-Objective Genetic Local Search Algorithm and Its Application to Flowshop Scheduling," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 28, no. 3, pp. 392–403, 1998.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function non_domination_levels(ys)
    L, m = 0, length(ys)
    levels = zeros(Int, m)
    while minimum(levels) == 0
        L += 1
        for (i,y) in enumerate(ys)
            if levels[i] == 0 &&
               !any((levels[i] == 0 || levels[i] == L) &&
                    dominates(ys[i],y) for i in 1 : m)
                levels[i] = L
            end
        end
    end
    return levels
end
```

Algorithm 15.5. A function for getting the nondomination levels of an array of multiobjective function evaluations, `ys`.

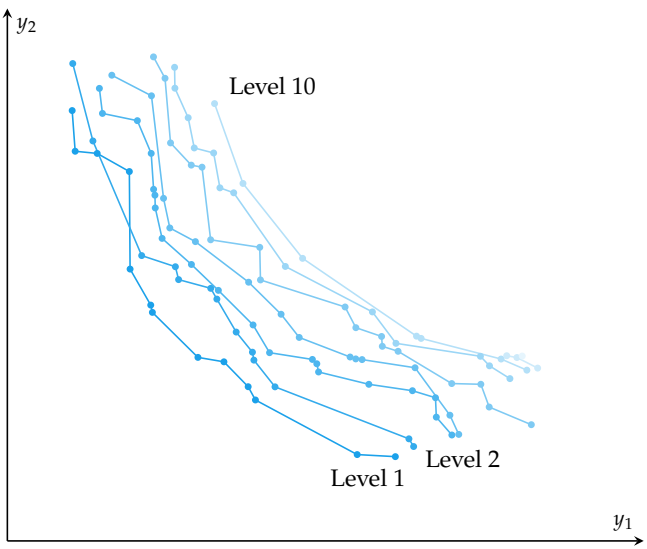


Figure 15.12. The nondomination levels for a population. Darker levels have lower (better) rankings.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

generation (algorithm 15.7). Individuals in the population that are not dominated by any individuals in the filter are added. Any dominated points in the filter are removed. Individuals from the Pareto filter can be injected into the population, thereby reducing the chance that portions of the Pareto frontier are lost between generations.

The filter often has a maximum capacity.[13] Filters that are overcapacity can be reduced by finding the closest pair of design points in the criterion space and removing one individual from that pair. This pruning method is implemented in algorithm 15.6. A Pareto filter obtained using a genetic algorithm is shown in figure 15.13.

[13] Typically the size of the population.

```
function discard_closest_pair!(xs, ys)
    index, min_dist = 0, Inf
    for (i,y) in enumerate(ys)
        for (j, y′) in enumerate(ys[i+1:end])
            dist = norm(y - y′)
            if dist < min_dist
                index, min_dist = rand([i,i+j]), dist
            end
        end
    end
    deleteat!(xs, index)
    deleteat!(ys, index)
    return (xs, ys)
end
```

Algorithm 15.6. The method `discard_closest_pair!` is used to remove one individual from a filter that is above capacity. The method takes the filter's list of design points `xs` and associated objective function values `ys`.

```
function update_pareto_filter!(filter_xs, filter_ys, xs, ys;
    capacity=length(xs))

    for (x,y) in zip(xs, ys)
        if !any(dominates(y′,y) for y′ in filter_ys)
            push!(filter_xs, x)
            push!(filter_ys, y)
        end
    end
    filter_xs, filter_ys = naive_pareto(filter_xs, filter_ys)
    while length(filter_xs) > capacity
        discard_closest_pair!(filter_xs, filter_ys)
    end
    return (filter_xs, filter_ys)
end
```

Algorithm 15.7. A method for updating a Pareto filter with design points `filter_xs`, corresponding objective function values `filter_ys`, a population with design points `xs` and objective function values `ys`, and filter capacity `capacity` that defaults to the population size.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

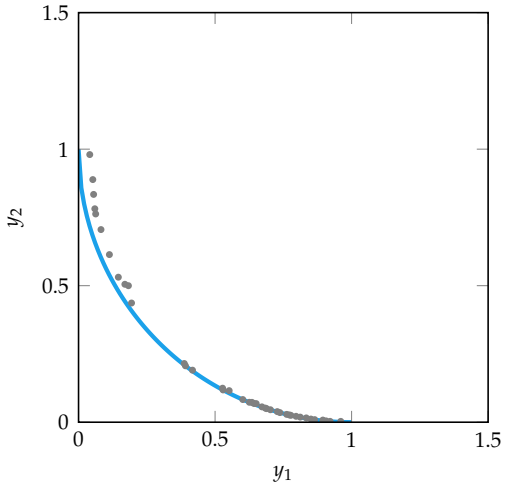


Figure 15.13. A Pareto filter used on the genetic algorithm in figure 15.11 to approximate the Pareto frontier.

15.4.4 Niche Techniques

The term *niche* refers to a focused cluster of points, typically in the criterion space, as shown in figure 15.14. Population methods can converge on a few niches, which limits their spread over the Pareto frontier. *Niche techniques* help encourage an even spread of points.

In *fitness sharing*,[14] shown in figure 15.15, an individual's objective values are penalized by a factor equal to the number of other points within a specified distance in the criterion space.[15] This scheme causes all points in a local region to share the fitness of the other points within the local region. Fitness sharing can be used together with nondomination ranking and subpopulation evaluation.

Equivalence class sharing can be applied to nondomination ranking. When comparing two individuals, the fitter individual is first determined based on the nondomination ranking. If they are equal, the better individual is the one with the fewest number of individuals within a specified distance in the criterion space.

Another niche technique has been proposed for genetic algorithms in which parents selected for crossover cannot be too close together in the criterion space. Selecting only nondominated individuals is also recommended.[16]

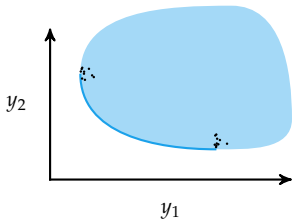


Figure 15.14. Two clear niches for a population in a two-dimensional criterion space.

[14] Fitness is inversely related to the objective being minimized.

[15] D. E. Goldberg and J. Richardson, "Genetic Algorithms with Sharing for Multimodal Function Optimization," in *International Conference on Genetic Algorithms*, 1987.

[16] S. Narayanan and S. Azarm, "On Improving Multiobjective Genetic Algorithms for Design Optimization," *Structural Optimization*, vol. 18, no. 2–3, pp. 146–155, 1999.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

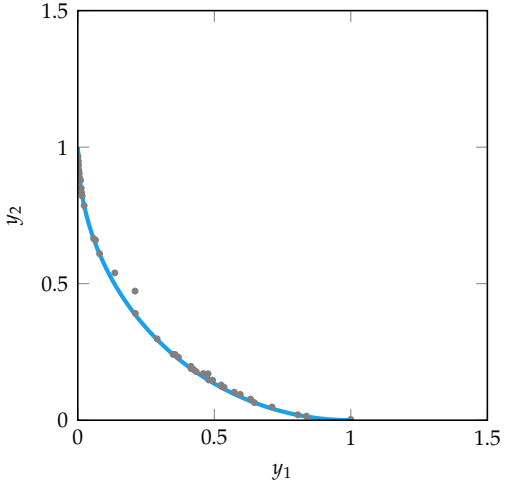


Figure 15.15. The results of applying fitness sharing to the Pareto filter in figure 15.13, thereby significantly improving its coverage.

15.5 Preference Elicitation

Preference elicitation involves inferring a scalar-valued objective function from preferences of experts about the trade-offs between objectives.[17] There are many different ways to represent the scalar-valued objective function, but this section will focus on the weighted sum model where $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{f}(\mathbf{x})$. Once we identify a suitable $\mathbf{w}$, we can use this scalar-valued objective function to find an optimal design.

15.5.1 Model Identification

A common approach for identifying the weight vector $\mathbf{w}$ in our preference model involves asking experts to state their preference between two points $\mathbf{a}$ and $\mathbf{b}$ in the criterion space $\mathcal{Y}$ (figure 15.16). Each of these points is the result of optimizing for a point on the Pareto frontier using associated weight vectors $\mathbf{w_a}$ and $\mathbf{w_b}$. The expert's response is either a preference for $\mathbf{a}$ or a preference for $\mathbf{b}$. There are other schemes for eliciting preference information, such as ranking points in the criterion space, but this binary preference query has been shown to pose minimal cognitive burden on the expert.[18]

[17] This section overviews non-Bayesian approaches to preference elicitation. For Bayesian approaches, see: S. Guo and S. Sanner, "Real-Time Multiattribute Bayesian Preference Elicitation with Pairwise Comparison Queries," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010. J. R. Lepird, M. P. Owen, and M. J. Kochenderfer, "Bayesian Preference Elicitation for Multiobjective Engineering Design Optimization," *Journal of Aerospace Information Systems*, vol. 12, no. 10, pp. 634–645, 2015.

[18] V. Conitzer, "Eliciting Single-Peaked Preferences Using Comparison Queries," *Journal of Artificial Intelligence Research*, vol. 35, pp. 161–191, 2009.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

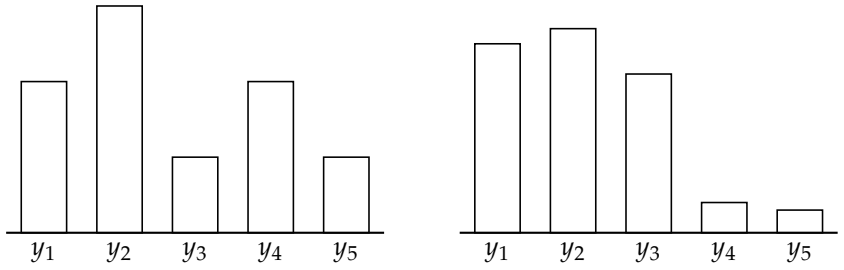


Figure 15.16. Preference elicitation schemes often involve asking experts their preferences between two points in the criterion space.

Suppose the outcomes of the expert queries have resulted in a set of criterion pairs

$$\left\{(\mathbf{a}^{(1)}, \mathbf{b}^{(1)}), \ldots, (\mathbf{a}^{(n)}, \mathbf{b}^{(n)})\right\} \tag{15.10}$$

where $\mathbf{a}^{(i)}$ is preferable to $\mathbf{b}^{(i)}$ in each pair. For each of these preferences, the weight vector must satisfy

$$\mathbf{w}^\top \mathbf{a}^{(i)} < \mathbf{w}^\top \mathbf{b}^{(i)} \implies (\mathbf{a}^{(i)} - \mathbf{b}^{(i)})^\top \mathbf{w} < 0 \tag{15.11}$$

In order to be consistent with the data, the weight vector must satisfy

$$\begin{cases} (\mathbf{a}^{(i)} - \mathbf{b}^{(i)})^\top \mathbf{w} < 0 \text{ for } i \text{ in } 1:n \\ \mathbf{1}^\top \mathbf{w} = 1 \\ \mathbf{w} \geq \mathbf{0} \end{cases} \tag{15.12}$$

Many different weight vectors could potentially satisfy the above equation. One approach is to choose a $\mathbf{w}$ that best separates $\mathbf{w}^\top \mathbf{a}^{(i)}$ from $\mathbf{w}^\top \mathbf{b}^{(i)}$

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize}} \quad & \sum_{i=1}^{n} (\mathbf{a}^{(i)} - \mathbf{b}^{(i)})^\top \mathbf{w} \\ \text{subject to} \quad & (\mathbf{a}^{(i)} - \mathbf{b}^{(i)})^\top \mathbf{w} < 0 \quad \text{for } i \text{ in } 1:n \\ & \mathbf{1}^\top \mathbf{w} = 1 \qquad \mathbf{w} \geq \mathbf{0} \end{aligned} \tag{15.13}$$

It is often desirable to choose the next weight vector such that it minimizes the distance from the previous weight vector. We can replace the objective function in equation (15.13) with $\|\mathbf{w} - \mathbf{w}^{(n)}\|_1$, thereby ensuring that our new weight vector $\mathbf{w}^{(n+1)}$ is as close as possible to our current one.[19]

[19] The previous weight vector may or may not be consistent with the added constraint $(\mathbf{a}^{(n)} - \mathbf{b}^{(n)})^\top \mathbf{w} < 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

15.5.2 *Paired Query Selection*

We generally want to choose the two points in the criterion space so that the outcome of the query is as informative as possible. There are many different approaches for such *paired query selection*, but we will focus on methods that attempt to reduce the space of weights consistent with *expert responses*, preference information supplied by a domain expert.

We will denote the set of weights consistent with expert responses as $\mathcal{W}$, which is defined by the linear constraints in equation (15.12). Because weights are bounded between 0 and 1, the feasible set is an enclosed region forming a convex polytope with finite volume. We generally want to reduce the volume of $\mathcal{W}$ in as few queries as possible.

Q-Eval[20], shown in figure 15.17, is a greedy elicitation strategy that heuristically seeks to reduce the volume of $\mathcal{W}$ as quickly as possible with each iteration. It chooses the query that comes closest to bisecting $\mathcal{W}$ into two equal parts. The method operates on a finite sampling of Pareto-optimal design points.

[20] V. S. Iyengar, J. Lee, and M. Campbell, "Q-EVAL: Evaluating Multiple Attribute Items Using Queries," in *ACM Conference on Electronic Commerce*, 2001.

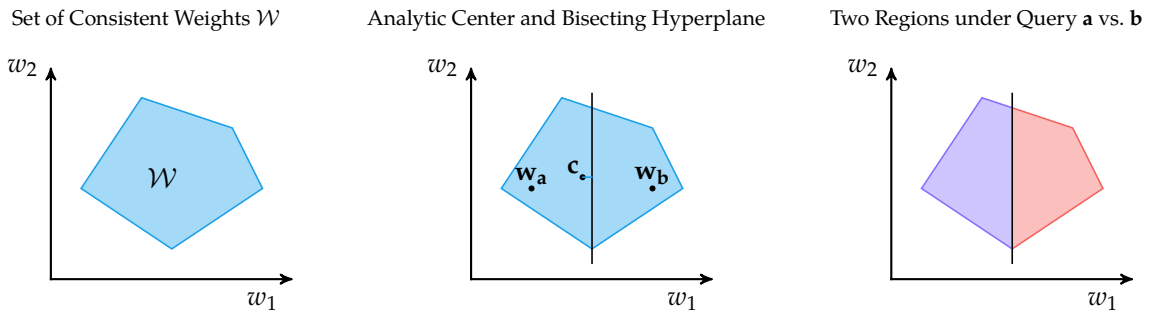


Figure 15.17. The Q-Eval greedy elicitation strategy. The figure shows the initial set of weights $\mathcal{W}$ consistent with previous preferences, a pair of weight vectors and their corresponding bisecting hyperplane, and the two polytopes formed by splitting along the bisecting hyperplane. The algorithm considers all possible pairs from a finite sampling of Pareto-optimal design points and chooses the query that most evenly splits $\mathcal{W}$.

The procedure for choosing a query pair in Q-Eval is as follows:

1. Compute the *prime analytic center* $\mathbf{c}$ of $\mathcal{W}$, which is the point that maximizes the sum of the logarithms of the distances between itself and the closest point on each nonredundant constraint in $\mathcal{W}$:

$$\mathbf{c} = \arg\max_{\mathbf{w} \in \mathcal{W}} \sum_{i=1}^{n} \ln\left((\mathbf{b}^{(i)} - \mathbf{a}^{(i)})^\top \mathbf{w}\right) \tag{15.14}$$

2. Compute the normal distance from the bisecting hyperplane between each pair of points and the center.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

3. Sort the design-point pairs in order of increasing distance.
4. For each of the k hyperplanes closest to $\mathbf{c}$, compute the volume ratio of the two polytopes formed by splitting $\mathcal{W}$ along the hyperplane.
5. Choose the design-point pair with split ratio closest to 1.

The *polyhedral method*[21] works by approximating $\mathcal{W}$ with a bounding ellipsoid centered at the analytic center of $\mathcal{W}$ as shown in figure 15.18. Queries are designed to partition the bounding ellipsoid into approximately equal parts and to favor cuts that are perpendicular to the longest axis of the ellipsoid both to reduce uncertainty and to balance the breadth in each dimension.

[21] D. Braziunas and C. Boutilier, "Elicitation of Factored Utilities," *AI Magazine*, vol. 29, no. 4, pp. 79–92, 2009.

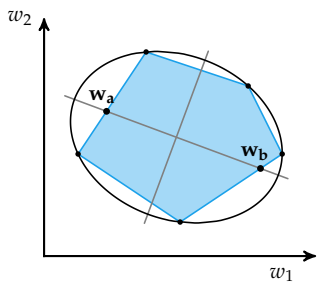


Figure 15.18. The polyhedral method uses a bounding ellipsoid for $\mathcal{W}$.

15.5.3 Design Selection

The previous section discussed query methods that select query pairs for efficiently reducing the search space. After query selection is complete, one must still select a final design. This process is known as *design selection*.

One such method, *decision quality improvement*,[22] is based on the idea that if we have to commit to a particular design, we should commit to the one for which the worst-case objective value is lowest:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{X}} \max_{\mathbf{w}\in\mathcal{W}} \mathbf{w}^\top \mathbf{f}(\mathbf{x}) \tag{15.15}$$

This *minimax decision* is robust because it provides an upper bound on the objective value.

The *minimax regret*[23] instead minimizes the maximum amount of regret the user can have when selecting a particular design:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{X}} \underbrace{\max_{\mathbf{w}\in\mathcal{W}} \max_{\mathbf{x}'\in\mathcal{X}} \mathbf{w}^\top \mathbf{f}(\mathbf{x}) - \mathbf{w}^\top \mathbf{f}(\mathbf{x}')}_{\text{maximum regret}} \tag{15.16}$$

where $\mathbf{w}^\top \mathbf{f}(\mathbf{x}) - \mathbf{w}^\top \mathbf{f}(\mathbf{x}')$ is the *regret* associated with choosing design $\mathbf{x}$ instead of design $\mathbf{x}'$ under the preference weight vector $\mathbf{w}$. Minimax regret can be viewed as accounting for the decision system's uncertainty with respect to the designer's true utility function.

The minimax regret can be used as a stopping criterion for preference elicitation strategies. We can terminate the preference elicitation procedure once the minimax regret drops below a certain threshold.

[22] D. Braziunas and C. Boutilier, "Minimax Regret-Based Elicitation of Generalized Additive Utilities," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007.

[23] C. Boutilier, R. Patrascu, P. Poupart, and D. Schuurmans, "Constraint-Based Optimization and Utility Elicitation Using the Minimax Decision Criterion," *Artificial Intelligence*, vol. 170, no. 8–9, pp. 686–713, 2006.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

15.6 Summary

- Design problems with multiple objectives often involve trading performance between different objectives.
- The Pareto frontier represents the set of potentially optimal solutions.
- Vector-valued objective functions can be converted to scalar-valued objective functions using constraint-based or weight-based methods.
- Population methods can be extended to produce individuals that span the Pareto frontier.
- Knowing the preferences of experts between pairs of points in the criterion space can help guide the inference of a scalar-valued objective function.

15.7 Exercises

Exercise 15.1. The weighted sum method is a very simple approach, and it is indeed used by engineers for multiobjective optimization. What is one disadvantage of the procedure when it is used to compute the Pareto frontier?

Solution: The weighted sum method cannot find Pareto-optimal points in nonconvex regions of the Pareto frontier.

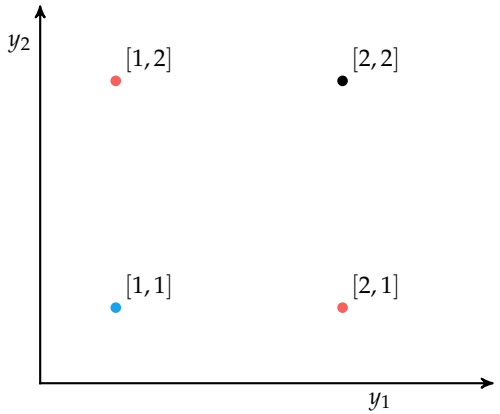
Exercise 15.2. Why are population methods well-suited for multiobjective optimization?

Solution: Nonpopulation methods will identify only a single point in the Pareto frontier. The Pareto frontier is very valuable in informing the designer of the trade-offs among a set of very good solutions. Population methods can spread out over the Pareto frontier and be used as an approximation of the Pareto frontier.

Exercise 15.3. Suppose you have the points $\{[1,2],[2,1],[2,2],[1,1]\}$ in the criterion space and you wish to approximate a Pareto frontier. Which points are Pareto optimal with respect to the rest of the points? Are there any weakly Pareto-optimal points?

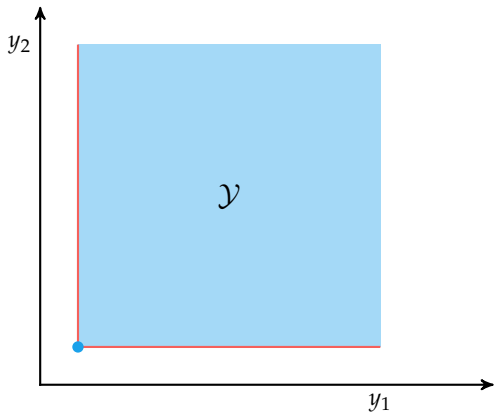
Solution: The only Pareto-optimal point is $[1,1]$. Both $[1,2]$ and $[2,1]$ are weakly Pareto optimal.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 15.4. Consider a square criterion space $\mathcal{Y}$ with $y_1 \in [0, 1]$ and $y_2 \in [0, 1]$. Plot the criterion space, indicate the Pareto-optimal points, and indicate the weakly Pareto optimal points.

Solution: The only Pareto-optimal point is $\mathbf{y} = [0, 0]$. The rest of the points on the bottom-left border are weakly Pareto-optimal.



Exercise 15.5. Enforcing $\mathbf{w} \geq \mathbf{0}$ and $\|\mathbf{w}\|_1 = 1$ in the weighted sum method is not sufficient for Pareto optimality. Give an example where coordinates with zero-valued weights find weakly Pareto-optimal points.

Solution: Consider the square criterion space from the previous question. Using $\mathbf{w} = [0, 1]$ assigns zero value to the first objective, causing the entire bottom edge of the criterion space to have equal value. As discussed above, only $\mathbf{y} = [0, 0]$ is Pareto optimal; the rest are weakly Pareto optimal.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 15.6. Provide an example where goal programming does not produce a Pareto-optimal point.

Solution: For example, if $\mathbf{y}^{\text{goal}}$ is in the criterion set, the goal programming objective will be minimized by $\mathbf{y}^{\text{goal}}$. If $\mathbf{y}^{\text{goal}}$ is also not Pareto optimal, the solution will not be Pareto optimal either.

Exercise 15.7. Use the constraint method to obtain the Pareto curve for the optimization problem:

$$\underset{x}{\text{minimize}} \;\; [x^2, (x-2)^2]$$

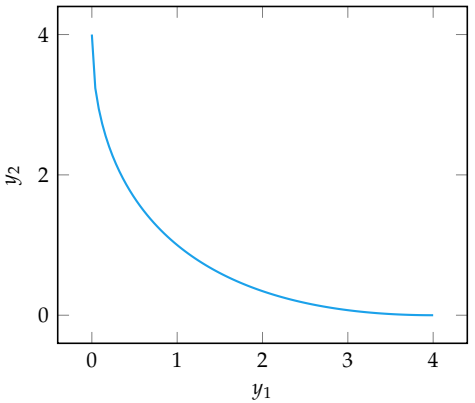
Solution: The constraint method constrains all but one objective. A Pareto curve can be generated by varying the constraints. If we constrain the first objective, each optimization problem has the form:

$$\begin{aligned} &\underset{x}{\text{minimize}} && (x-2)^2 \\ &\text{subject to} && x^2 \leq c \end{aligned}$$

The constraint can be satisfied only for $c \geq 0$. This allows x to vary between $\pm\sqrt{c}$. The second objective is optimized by minimizing the deviation of x from 2. Thus, for a given value of c, we obtain:

$$x^* = \begin{cases} 2 & \text{if } c \geq 4 \\ \sqrt{c} & \text{if } c \in [0, 4) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting Pareto curve is:



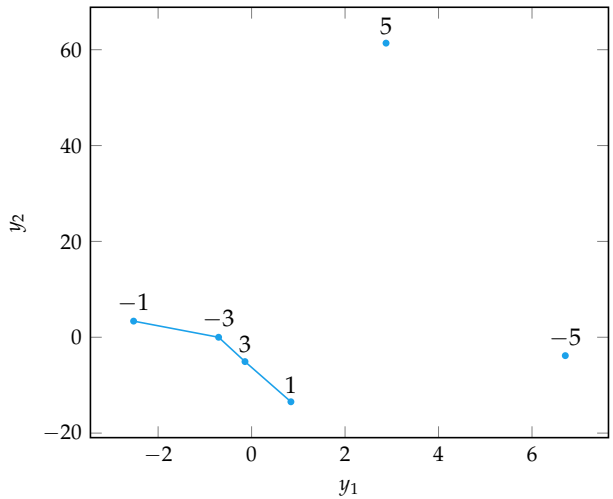
Exercise 15.8. Suppose we have a multiobjective optimization problem where the two objectives are as follows:

$$f_1(x) = -(x-2)\sin(x)$$
$$f_2(x) = -(x+3)^2\sin(x)$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

With $x \in \{-5, -3, -1, 1, 3, 5\}$, plot the points in the criterion space. How many points are on the Pareto frontier?

Solution: The criterion space is the space of objective function values. The resulting plot is:



We find that four points are on the approximate Pareto frontier corresponding to our six sample points. The corresponding design points are $x = \{-1, -3, 3, 1\}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

16 Sampling Plans

For many optimization problems, function evaluations can be quite expensive. For example, evaluating a hardware design may require a lengthy fabrication process, an aircraft design may require a wind tunnel test, and new deep learning hyperparameters may require a week of GPU training. A common approach for optimizing in contexts where evaluating design points is expensive is to build a *surrogate model*, which is a model of the optimization problem that can be efficiently optimized in lieu of the true objective function. Further evaluations of the true objective function can be used to improve the model. Fitting such models requires an initial set of points, ideally points that are *space-filling*; that is, points that cover the region as well as possible. This chapter covers different *sampling plans* for covering the search space when we have limited resources.[1]

[1] There are other references that discuss the topics in this chapter in greater detail. See, for example: G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, 2nd ed. Wiley, 2005. A. Dean, D. Voss, and D. Draguljić, *Design and Analysis of Experiments*, 2nd ed. Springer, 2017. D. C. Montgomery, *Design and Analysis of Experiments*. Wiley, 2017.

16.1 Full Factorial

The *full factorial* sampling plan (algorithm 16.1) places a grid of evenly spaced points over the search space. This approach is easy to implement, does not rely on randomness, and covers the space, but it uses a large number of points. A grid of evenly spaced points is spread over the search space as shown in figure 16.1. Optimization over the points in a full factorial sampling plan is referred to as *grid search*.

The sampling grid is defined by a lower-bound vector $\mathbf{a}$ and an upper-bound vector $\mathbf{b}$ such that $a_i \leq x_i \leq b_i$ for each component i. For a grid with m_i samples in the ith dimension, the nearest points are separated by a distance $(b_i - a_i)/(m_i - 1)$.

The full factorial method requires a sample count exponential in the number of dimensions.[2] For n dimensions with m samples per dimension, we have m^n

[2] The full factorial method gets its name not from a factorial sample count (it is exponential) but from designing with two or more *discrete factors*. Here the factors are the m discretized levels associated with each variable.

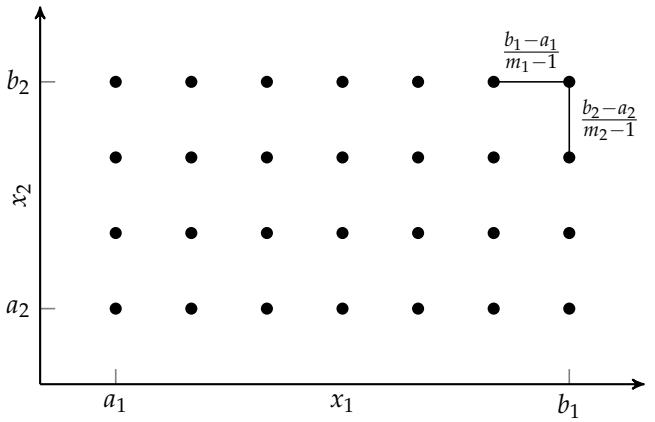


Figure 16.1. Full factorial search covers the search space in a grid of points.

total samples. This exponential growth is far too high to be of practical use when there are more than a few variables. Even when full factorial sampling is able to be used, the grid points are generally forced to be quite coarse and therefore can easily miss small, local features of the optimization landscape.

```
function samples_full_factorial(a, b, m)
    ranges = [range(a[i], stop=b[i], length=m[i])
              for i in eachindex(a)]
    return collect.(collect(Iterators.product(ranges...)))
end
```

Algorithm 16.1. A function for obtaining all sample locations for the full factorial grid. Here, `a` is a vector of variable lower bounds, `b` is a vector of variable upper bounds, and `m` is a vector of sample counts for each dimension.

16.2 Random Sampling

A straightforward alternative to full factorial sampling is *random sampling*, which simply draws m random samples over the design space using a pseudo-random number generator. To generate a random sample $\mathbf{x}$, we can sample each variable independently from a distribution. If we have bounds on the variables, such as $a_i \leq x_i \leq b_i$, a common approach is to use a uniform distribution over $[a_i, b_i]$, although other distributions may be used. For some variables, it may make sense to use a log-uniform distribution.[3] The samples of design points are uncorrelated with each other. The hope is that the randomness, on its own, will result in an adequate cover of the design space.

[3] Some parameters, such as the step factor for deep neural networks, are best searched in log-space.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

16.3 Uniform Projection Plans

Suppose we have a two-dimensional optimization problem discretized into an $m \times m$ sampling grid as with the full factorial method, but, instead of taking all m^2 samples, we want to sample only m positions. We could choose the samples at random, but not all arrangements are equally useful. We want the samples to be spread across the space, and we want the samples to be spread across each individual component.

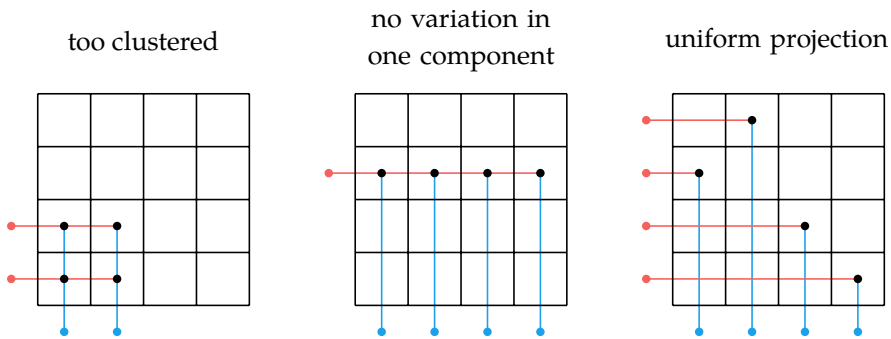


Figure 16.2. Several ways to choose m samples from a two-dimensional grid. We generally prefer sampling plans that cover the space and vary across each component.

A *uniform projection plan* is a sampling plan over a discrete grid where the distribution over each dimension is uniform. For example, in the rightmost sampling plan in figure 16.2, each row has exactly one entry and each column has exactly one entry.

A uniform projection plan with m samples on an $m \times m$ grid can be constructed using an m-element permutation as shown in figure 16.3. There are therefore $m!$ possible uniform projection plans. This set of possible projection plans grows quickly. Even for $m = 5$, this is already $5! = 120$ possible plans. For $m = 10$, there are 3,628,800 plans.

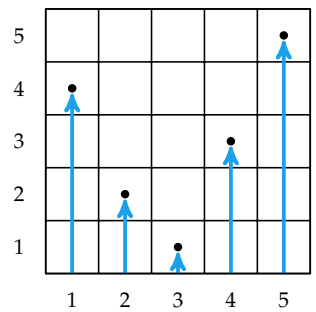


Figure 16.3. Constructing a uniform projection plan using the permutation $p = [4, 2, 1, 3, 5]$.

Sampling with uniform projection plans is sometimes called *Latin-hypercube sampling* because of the connection to *Latin squares* (figure 16.4). A Latin square is an $m \times m$ grid where each row contains each integer 1 through m and each column contains each integer 1 through m. Latin-hypercubes are a generalization to any number of dimensions.

4	1	3	2
1	4	2	3
3	2	1	4
2	3	4	1

Figure 16.4. A 4×4 Latin square. A uniform projection plan can be constructed by choosing a value $i \in \{1, 2, 3, 4\}$ and sampling all cells with that value.

Uniform projection plans for n dimensions can be constructed using a permutation for each dimension (algorithm 16.2).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function uniform_projection_plan(m, n)
    perms = [randperm(m) for i in 1 : n]
    [[perms[i][j] for i in 1 : n] for j in 1 : m]
end
```

Algorithm 16.2. A function for constructing a uniform projection plan for an `n`-dimensional hypercube with `m` samples per dimension. It returns a vector of index vectors.

16.4 Stratified Sampling

Many sampling plans, including uniform projection and full factorial plans, are based on an $m \times m$ grid. Such a grid, even if fully sampled, could miss important information due to systematic regularities as shown in figure 16.5. One method for providing an opportunity to hit every point is to use *stratified sampling*.

Stratified sampling modifies any grid-based sampling plan, including full factorial and uniform projection plans. Cells are sampled at a point chosen uniformly at random from within the cell rather than at the cell's center as shown in figure 16.6.

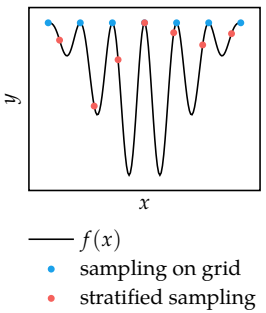


Figure 16.5. Using an evenly spaced grid on a function with systematic regularities can miss important information.

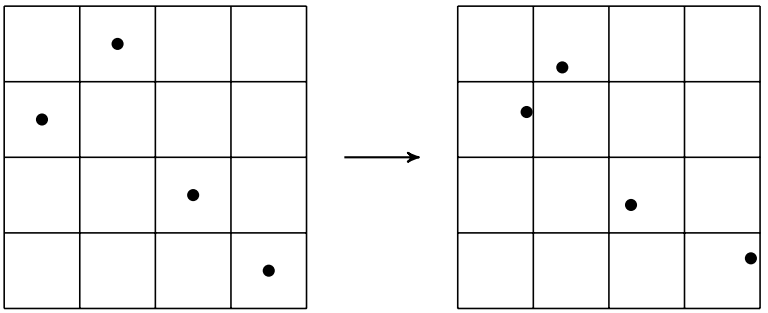


Figure 16.6. Stratified sampling applied to a uniform projection plan.

16.5 Space-Filling Metrics

A good sampling plan fills the design space since the ability for a surrogate model to generalize from samples decays with the distance from those samples. Not all plans, even uniform projection plans, are equally good at covering the search space. For example, a grid diagonal (figure 16.7) is a uniform projection plan but only covers a narrow strip. This section discusses different *space-filling metrics* for measuring the degree to which a sampling plan $X \subseteq \mathcal{X}$ fills the design space.

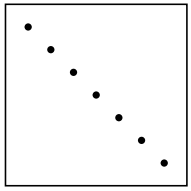


Figure 16.7. A uniform projection plan that is not space-filling.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

16.5.1 Discrepancy

The ability of the sampling plan to fill a hyperrectangular design space $\mathcal{H}$ can be measured by its *discrepancy*.[4] If X has low discrepancy, then a randomly chosen subset of the design space should contain a fraction of samples proportional to the subset's volume relative to that of $\mathcal{H}$.[5] The discrepancy associated with X is the maximum difference between the fraction of samples in a hyperrectangular subset H and the fraction of the subset's volume with respect to the volume of the design space:

$$d(X) = \operatorname*{supremum}_{H \subseteq \mathcal{H}} \left| \frac{\#(X \bigcap H)}{\#(X \bigcap \mathcal{H})} - \frac{\lambda(H)}{\lambda(\mathcal{H})} \right| \tag{16.1}$$

where $\#(X \bigcap H)$ is the number of points in X that lie in the hyperrectangle H. Note that $\#(X \bigcap \mathcal{H})$ will be the number of points in X, as all points lie in the design space. The value $\lambda(H)$ is the n-dimensional volume of the given hyperrectangle, which is the product of its side lengths. The *supremum* of a set of values is similar to the maximum except that the maximum value must occur in the set, while the supremum might only be approached by values in the set, as shown in example 16.1.[6] Computing the discrepancy in this way for hyperrectangles can be difficult, and it can be even less straightforward for non-hyperrectangles.

[4] L. Kuipers and H. Niederreiter, *Uniform Distribution of Sequences*. Dover, 2012.

[5] In arbitrary dimensions, we can use the *Lebesgue measure*, which is a generalization of volume to any subset of n-dimensional Euclidean space. It is length in one-dimensional space, area in two-dimensional space, and volume in three-dimensional space.

[6] This definition of discrepancy requires hyperrectangles. The notion of discrepancy can be extended to allow $\mathcal{H}$ to include other sets, such as convex polytopes.

16.5.2 Pairwise Distances

An alternative method for measuring how space-filling a sampling plan is involves analyzing the *pairwise distances* between all points. Algorithm 16.3 computes the pairwise distances between all points in a sampling plan. Sampling plans that are more space-filling will tend to have larger pairwise distances.

We can compare two different sampling plans by sorting each set's pairwise distances in ascending order. As implemented in algorithm 16.4, the plan with the first pairwise distance that exceeds the other is considered more space-filling.

There are several methods for generating space-filling uniform projection plans. One method simply generates candidates at random and then uses the one that is most space-filling. Alternatively, we can produce candidates by repeatedly mutating a uniform projection plan in a way that preserves the uniform projection property (algorithm 16.5). We can also incorporate simulated annealing to search the space of sampling plans.

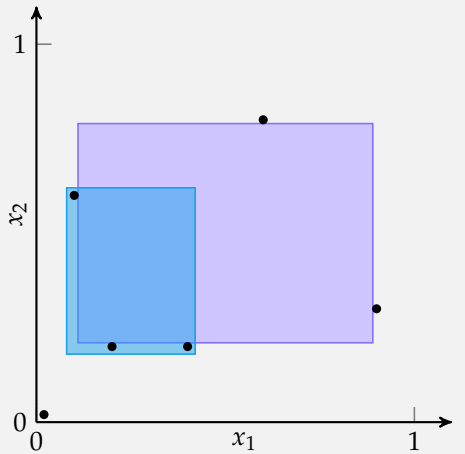
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 16.1. Computing the discrepancy for a sampling plan over the unit square. The sizes of the rectangles are slightly exaggerated to clearly show which points they contain.

Consider the set:

$$X = \left\{ \left[\frac{1}{5}, \frac{1}{5}\right], \left[\frac{2}{5}, \frac{1}{5}\right], \left[\frac{1}{10}, \frac{3}{5}\right], \left[\frac{9}{10}, \frac{3}{10}\right], \left[\frac{1}{50}, \frac{1}{50}\right], \left[\frac{3}{5}, \frac{4}{5}\right] \right\}$$

The discrepancy of X with respect to the unit square is determined by a rectangular subset H that either has very small area but contains very many points or has very large area and contains very few points.



The blue rectangle, $x_1 \in \left[\frac{1}{10}, \frac{2}{5}\right]$, $x_2 \in \left[\frac{1}{5}, \frac{3}{5}\right]$, has a volume of 0.12 and contains 3 points. Its corresponding discrepancy measure is thus 0.38.

The purple rectangle, $x_1 \in \left[\frac{1}{10} + \epsilon, \frac{9}{10} - \epsilon\right]$, $x_2 \in \left[\frac{1}{5} + \epsilon, \frac{4}{5} - \epsilon\right]$, produces an even higher discrepancy. As ϵ approaches zero, the volume and the discrepancy approach 0.48 because the rectangle contains no points. Note that the limit was required, reflecting the need to use a supremum in the definition of discrepancy.

```
import LinearAlgebra: norm
function pairwise_distances(X, p=2)
    m = length(X)
    [norm(X[i]-X[j], p) for i in 1:(m-1) for j in (i+1):m]
end
```

Algorithm 16.3. A function for obtaining the list of pairwise distances between points in sampling plan `X` using the L_p norm specified by `p`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function compare_sampling_plans(A, B, p=2)
    pA = sort(pairwise_distances(A, p))
    pB = sort(pairwise_distances(B, p))
    for (dA, dB) in zip(pA, pB)
        if dA < dB
            return 1
        elseif dA > dB
            return -1
        end
    end
    return 0
end
```

Algorithm 16.4. A function for comparing the degree to which two sampling plans `A` and `B` are space-filling using the L_p norm specified by `p`. The function returns `-1` if `A` is more space-filling than `B`. It returns `1` if `B` is more space-filling than `A`. It returns `0` if they are equivalent.

```
function mutate!(X)
    m, n = length(X), length(X[1])
    j = rand(1:n)
    i = randperm(m)[1:2]
    X[i[1]][j], X[i[2]][j] = X[i[2]][j], X[i[1]][j]
    return X
end
```

Algorithm 16.5. A function for mutating uniform projection plan `X`, while maintaining its uniform projection property.

16.5.3 *Morris-Mitchell Criterion*

The comparison scheme in section 16.5.2 typically results in a challenging optimization problem with many local minima. An alternative is to search for a plan that minimizes the *Morris-Mitchell criterion* (algorithm 16.6):[7]

$$\Phi_q(X) = \left(\sum_i d_i^{-q}\right)^{1/q} \tag{16.2}$$

where d_i is the ith pairwise distance between points in X and $q > 0$ is a tunable parameter.[8] Morris and Mitchell recommend optimizing:

$$\underset{X}{\text{minimize}} \underset{q \in \{1,2,5,10,20,50,100\}}{\text{maximize}} \Phi_q(X) \tag{16.3}$$

[7] M. D. Morris and T. J. Mitchell, "Exploratory Designs for Computational Experiments," *Journal of Statistical Planning and Inference*, vol. 43, no. 3, pp. 381–402, 1995.

[8] We can view $\Phi_q(X)$ as the application of the L_q norm to a vector containing inverse pairwise distances from the points in X.

```
function phiq(X, q=1, p=2)
    dists = pairwise_distances(X, p)
    return sum(dists.^(-q))^(1/q)
end
```

Algorithm 16.6. An implementation of the Morris-Mitchell criterion that takes a list of design points `X`, the criterion parameter `q > 0`, and a norm parameter `p ≥ 1`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Figure 16.8 shows the Morris-Mitchell criterion evaluated for several uniform projection plans.

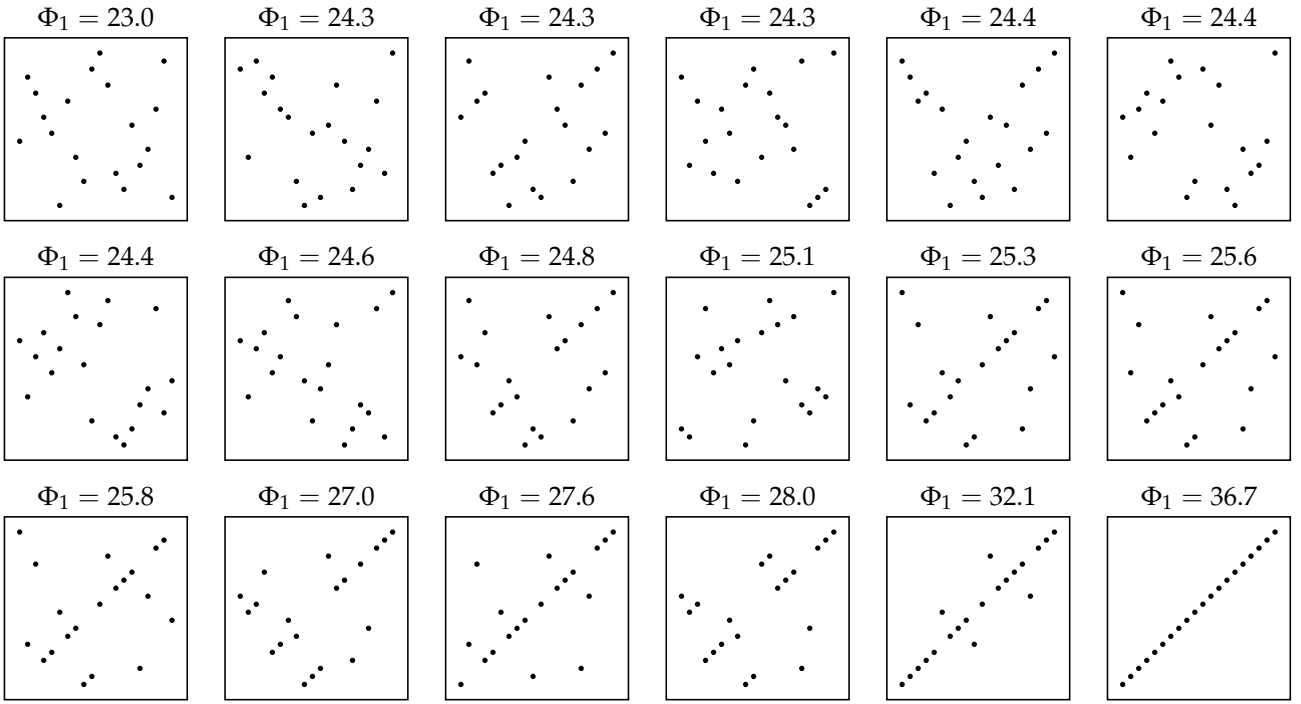


Figure 16.8. Uniform projection plans sorted from best to worst according to Φ_1.

16.6 Space-Filling Subsets

In some cases, we have a set of points X and want to find a subset of points $S \subset X$ that still maximally fills X. The need for identifying *space-filling subsets* of X arises in the context of multifidelity models.[9] For example, suppose we used a sampling plan X to identify a variety of aircraft wing designs to evaluate using computational fluid dynamic models in simulation. We can choose only a subset of these design points S to build and test in a wind tunnel. We still want S to be space filling.

[9] A. I. J. Forrester, A. Sóbester, and A. J. Keane, "Multi-Fidelity Optimization via Surrogate Modelling," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 463, no. 2088, pp. 3251–3269, 2007.

The degree to which S fills the design space can be quantified using the maximum distance between a point in X and the closest point in S. This metric generalizes to any two finite sets A and B (algorithm 16.7). We can use any L_p

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

norm, but we typically use L_2, the Euclidean distance:

$$d_{\max}(X, S) = \underset{\mathbf{x} \in X}{\text{maximize}} \, \underset{\mathbf{s} \in S}{\text{minimize}} \|\mathbf{s} - \mathbf{x}\|_p \tag{16.4}$$

```
min_dist(a, B, p) = minimum(norm(a-b, p) for b in B)
d_max(A, B, p=2) = maximum(min_dist(a, B, p) for a in A)
```

Algorithm 16.7. The set L_p distance metrics between two discrete sets, where `A` and `B` are lists of design points and `p` is the L_p norm parameter.

A space-filling sampling plan is one that minimizes this metric.[10] Finding a space-filling sampling plan with m elements is an optimization problem

$$\begin{aligned} \underset{S}{\text{minimize}} \quad & d_{\max}(X, S) \\ \text{subject to} \quad & S \subseteq X \\ & \#S = m \end{aligned} \tag{16.5}$$

[10] We can also minimize the Morris-Mitchell criterion for S.

Optimizing equation (16.5) is typically computationally intractable. A brute force approach would try all $d!/(m!(d-m)!)$ size-m subsets for a dataset of d design points. There are various heuristic strategies that are more efficient but may not be optimal.

- In *greedy subset selection* (algorithm 16.8), we initialize S to contain a random point from X. We then add the next best point that minimizes the distance metric. We continue adding points until S is of the desired size.
- In the *exchange subset selection* (algorithm 16.9), we initialize S to a random subset of X. We repeatedly replace points that are in S with a different point in X that is not already in S to improve on the distance metric. We stop when we can no longer find an exchange that improves the metric.

Because both of these algorithms are initialized randomly, better results might be obtained through random restarts (section 4.6) and returning the best sampling plan. Figure 16.9 compares space-filling subsets obtained using greedy subset selection and exchange subset selection.

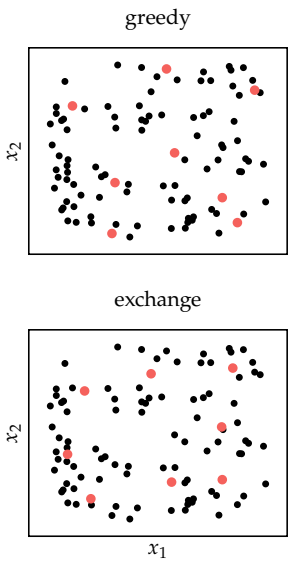


Figure 16.9. Space-filling subsets obtained with both greedy subset selection and exchange subset selection.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function greedy_subset_selection(X, m, d=d_max)
    S = [sample(X)]
    for i in 2 : m
        x = argmin(x->d(X, vcat(S, [x])), setdiff(X, S))
        push!(S, x)
    end
    return S
end
```

Algorithm 16.8. Greedy subset selection for finding `m`-element sampling plans that minimize a distance metric `d` for discrete set `X`.

```
function exchange_subset_selection(X, m, d=d_max)
    S = sample(X, m, replace=false)
    δ, done = d(X, S), false
    while !done
        best_pair = (0,0)
        for i in 1 : m
            s = S[i]
            for (j,x) in enumerate(X)
                if !(x in S)
                    S[i] = x
                    δ′ = d(X, S)
                    if δ′ < δ
                        δ = δ′
                        best_pair = (i,j)
                    end
                    S[i] = s
                end
            end
        end
        done = best_pair == (0,0)
        if !done
            i,j = best_pair
            S[i] = X[j]
        end
    end
    return S
end
```

Algorithm 16.9. Exchange subset selection for finding `m`-element sampling plans that minimize a distance metric `d` for discrete set `X`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

16.7 Quasi-Random Sequences

Quasi-random sequences,[11] also called *low-discrepancy sequences*, are often used in the context of trying to approximate an integral over a multidimensional space:

$$\int_{\mathcal{X}} f(\mathbf{x})\, d\mathbf{x} \approx \frac{v}{m} \sum_{i=1}^{m} f(\mathbf{x}^{(i)}) \tag{16.6}$$

where each $\mathbf{x}^{(i)}$ is sampled uniformly at random over the domain $\mathcal{X}$ and v is the volume of $\mathcal{X}$. This approximation is known as *Monte Carlo integration*.

Rather than relying on random or pseudo-random numbers to generate integration points, quasi-random sequences are deterministic sequences that fill the space in a systematic manner so that the integral converges as fast as possible in the number of points m. These *quasi-Monte Carlo methods* have an error convergence of $O(1/m)$ as opposed to $O(1/\sqrt{m})$ for typical Monte Carlo integration, as shown in figure 16.10.

Quasi-random sequences are typically constructed for the unit n-dimensional hypercube, $[0,1]^n$. Any multidimensional function with bounds on each variable can be transformed into such a hypercube.[12] There are various methods for generating quasi-random sequences. Figure 16.13 compares several methods against random sampling.

[11] C. Lemieux, *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer, 2009.

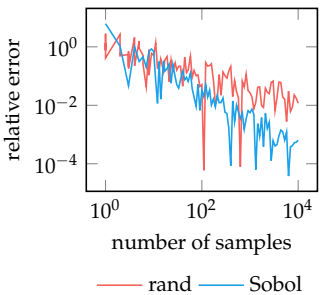


Figure 16.10. The error from estimating $\int_0^1 \sin(10x)\, dx$ using Monte Carlo integration with random numbers from $\mathcal{U}(0,1)$ and a Sobol sequence. The Sobol sequence, covered in section 16.7.3, converges faster.

[12] In Julia, a transformation from a hyperrectangle with lower bounds `a` and upper bounds `b` to a unit hypercube is given by `x->f((x-a)./(b-a))`.

16.7.1 Additive Recurrence

Simple recurrence relations of the form:

$$x^{(k+1)} = x^{(k)} + c \pmod 1 \tag{16.7}$$

produce space-filling sets provided that c is irrational. The value of c leading to the smallest discrepancy is

$$c = \varphi - 1 = \frac{\sqrt{5}-1}{2} \approx 0.618034 \tag{16.8}$$

where φ is the golden ratio.[13]

We can construct a space-filling set over n dimensions using an *additive recurrence* sequence for each coordinate, each with its own value of c. The square roots of the primes are known to be irrational, and can thus be used to obtain different sequences for each coordinate:

$$c_1 = \sqrt{2}, \quad c_2 = \sqrt{3}, \quad c_3 = \sqrt{5}, \quad c_4 = \sqrt{7}, \quad c_5 = \sqrt{11}, \quad \ldots \tag{16.9}$$

[13] C. Schretter, L. Kobbelt, and P.-O. Dehaye, "Golden Ratio Sequences for Low-Discrepancy Sampling," *Journal of Graphics Tools*, vol. 16, no. 2, pp. 95–104, 2016.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Methods for additive recurrence are implemented in algorithm 16.10.

```
function filling_set_additive_recurrence(m; c=φ-1)
    x0 = rand()
    return [mod(x0 + (k-1)*c, 1) for k in 1:m]
end
function filling_set_additive_recurrence(m, n)
    seqs = [filling_set_additive_recurrence(m, c=sqrt(prime(i)))
            for i in 1:n]
    return [collect(x) for x in zip(seqs...)]
end
```

Algorithm 16.10. Additive recurrence for constructing `m`-element filling sequences over `n`-dimensional unit hypercubes. The `prime(i)` method from `Primes.jl` produces the `i`th prime number.

16.7.2 *Halton Sequence*

The *Halton sequence* is a multidimensional quasi-random space-filling set.[14] The single-dimensional version, called *van der Corput sequences*, generates sequences where the unit interval is divided into powers of base b. For example, $b = 2$ produces:

$$X = \left\{\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \frac{1}{16}, \ldots\right\} \tag{16.10}$$

whereas $b = 5$ produces:

$$X = \left\{\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{1}{25}, \frac{6}{25}, \frac{11}{25}, \ldots\right\} \tag{16.11}$$

The values for $b = 2$ are shown in figure 16.11.

[14] J. H. Halton, "Algorithm 247: Radical-Inverse Quasi-Random Point Sequence," *Communications of the ACM*, vol. 7, no. 12, pp. 701–702, 1964.

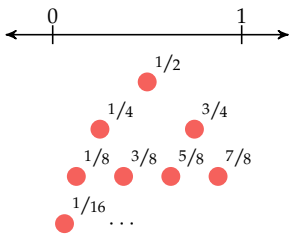


Figure 16.11. The first 8 values of the Halton sequence for $b = 2$.

```
function halton(i, b)
    result, f = 0.0, 1.0
    while i > 0
        f = f / b;
        result = result + f * mod(i, b)
        i = floor(Int, i / b)
    end
    return result
end
filling_set_halton(m; b=2) = [halton(i,b) for i in 1: m]
function filling_set_halton(m, n)
    seqs = [filling_set_halton(m, b=prime(i)) for i in 1:n]
    return [collect(x) for x in zip(seqs...)]
end
```

Algorithm 16.11. Halton quasi-random `m`-element filling sequences over `n`-dimensional unit hypercubes, where `b` is the base. The bases must be coprime.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Multidimensional space-filling sequences use one van der Corput sequence for each dimension, each with its own base b. In order to be uncorrelated, the bases must be *coprime*, meaning that the only positive integer that divides them both is 1. Methods for constructing Halton sequences are implemented in algorithm 16.11.

For large primes, we can get correlation in the first few numbers. Such a correlation is shown in figure 16.12. Correlation can be avoided by the *leaped Halton method*,[15] which takes every pth point, where p is a prime different from all coordinate bases.

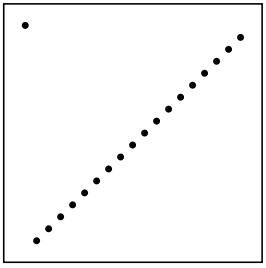


Figure 16.12. The Halton sequence with $\mathbf{b} = [19, 23]$ for which the first 18 samples are perfectly linearly correlated.

16.7.3 Sobol Sequences

Sobol sequences are quasi-random space-filling sequences for n-dimensional hypercubes.[16] They are generated by xor-ing the previous Sobol number with a set of direction numbers:

$$X_j^{(i)} = X_j^{(i-1)} \veebar v_j^{(k)} \tag{16.12}$$

where $v_j^{(k)}$ is the jth bit of the kth direction number. The symbol $\veebar$ denotes the xor operation, which returns true if and only if both inputs are different. Tables of good direction numbers have been provided by various authors.[17]

A comparison of these and previous approaches is shown in figure 16.13. Some methods exhibit a clear underlying structure.

16.8 Summary

- Sampling plans are used to cover search spaces with a limited number of points.
- Full factorial sampling, which involves sampling at the vertices of a uniformly discretized grid, requires a number of points exponential in the number of dimensions.
- Uniform projection plans, which project uniformly over each dimension, can be efficiently generated and can be optimized to be space filling.
- Greedy local search and the exchange algorithm can be used to find a subset of points that maximally fill a space.
- Quasi-random sequences are deterministic procedures by which space-filling sampling plans can be generated.

[15] L. Kocis and W. J. Whiten, "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, vol. 23, no. 2, pp. 266–294, 1997.

[16] Named for the Russian mathematician Ilya Meyerovich Sobol (1926–). I. M. Sobol, "On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, 1967.

[17] The `Sobol.jl` package provides an implementation for up to 21,201 dimensions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

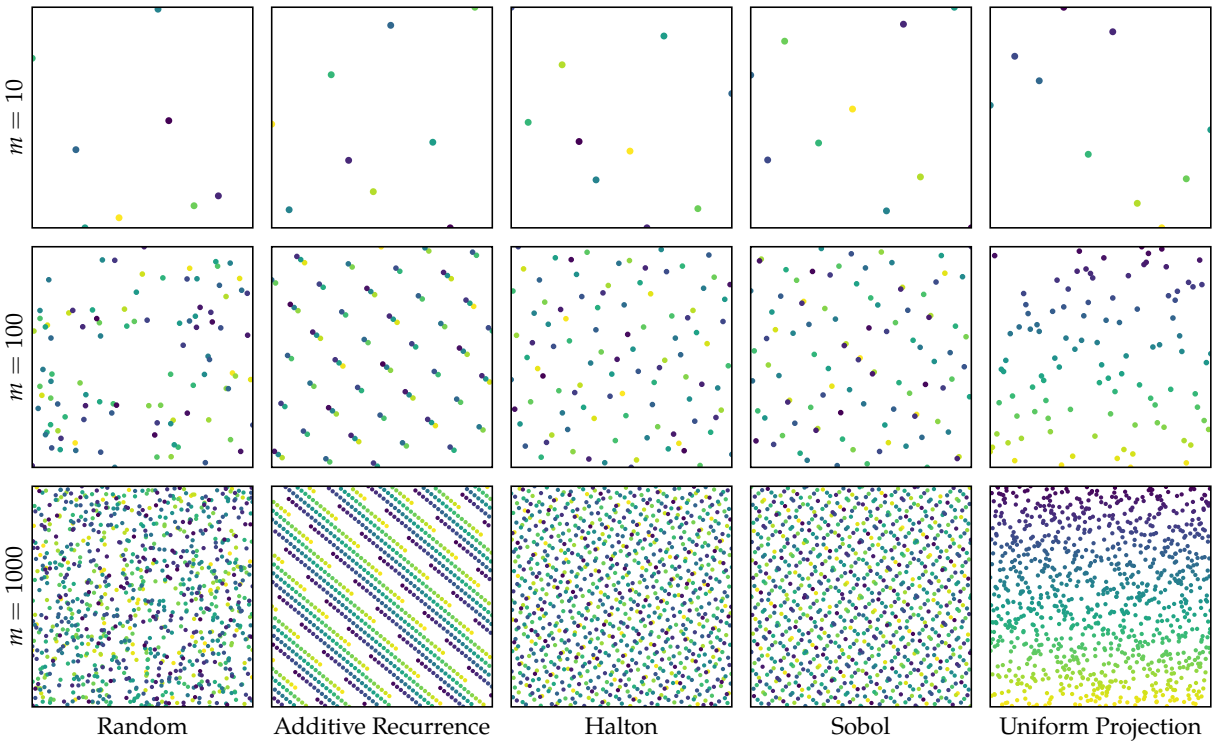


Figure 16.13. Space-filling sampling plans in two dimensions. Samples are colored according to the order in which they are sampled. The uniform projection plan was generated randomly.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

16.9 Exercises

Exercise 16.1. Filling a multidimensional space requires exponentially more points as the number of dimensions increases. To help build this intuition, determine the side lengths of an n-dimensional hypercube such that it fills half of the volume of the n-dimensional unit hypercube.

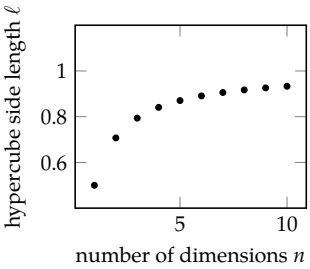
Solution: The one-dimensional unit hypercube is $x \in [0, 1]$, and its volume is 1. In this case the required side length ℓ is 0.5. The two-dimensional unit hypercube is the unit square $x_i \in [0, 1]$ for $i \in \{1, 2\}$, which has a two-dimensional volume, or area, of 1. The area of a square with side length ℓ is ℓ^2, so we solve:

$$\ell^2 = \frac{1}{2} \quad \implies \quad \ell = \frac{\sqrt{2}}{2} \approx 0.707$$

An n-dimensional hypercube has volume ℓ^n. We thus solve:

$$\ell^n = \frac{1}{2} \quad \implies \quad \ell = 2^{-1/n}$$

The side length approaches one.



Exercise 16.2. Suppose that you sample randomly inside a unit sphere in n dimensions. Compute the probability that a randomly sampled point is within ϵ distance from the surface of the sphere as $n \to \infty$. Hint: The volume of a sphere is $C(n)r^n$, where r is the radius and $C(n)$ is a function of the dimension n only.

Solution: The probability that a randomly sampled point is within ϵ-distance from the surface is just the ratio of the volumes. Thus:

$$P(\|x\|_2 > 1 - \epsilon) = 1 - P(\|x\|_2 < 1 - \epsilon) = 1 - (1 - \epsilon)^n \to 1$$

as $n \to \infty$.

Exercise 16.3. We have a sampling plan $X = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(10)}\}$, where

$$\mathbf{x}^{(i)} = [\cos(2\pi i/10), \sin(2\pi i/10)]$$

Suppose we use the Morris-Mitchell criterion with an L_2 norm with the parameter q set to 2. In other words, we want to evaluate $\Phi_2(X)$. If we add $[2, 3]$ to each $\mathbf{x}^{(i)}$, will $\Phi_2(X)$ change? Why or why not?

Solution: No. The Morris-Mitchell criterion is based entirely on pairwise distances. Shifting all of the points by the same amount does not change the pairwise distances and thus will not change $\Phi_2(X)$.

Exercise 16.4. Additive recurrence requires that the additive constant c in equation (16.7) be irrational. Why can c not be rational?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: A rational number can be written as a fraction of two integers a/b. It follows that the sequence repeats every b iterations:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \frac{a}{b} \pmod 1 \\ x^{(k)} &= x^{(0)} + k\frac{a}{b} \pmod 1 \\ &= x^{(0)} + k\frac{a}{b} + a \pmod 1 \\ &= x^{(0)} + (k+b)\frac{a}{b} \pmod 1 \\ &= x^{(k+b)} \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

17 Surrogate Models

The previous chapter discussed methods for producing a sampling plan. This chapter shows how to use these samples to construct models of the objective function that can be used in place of the real objective function. Such *surrogate models* are designed to be smooth and inexpensive to evaluate so that they can be efficiently optimized. The surrogate model can then be used to help direct the search for the optimum of the real objective function.[1]

[1] Other texts provide more extensive coverage of surrogate modeling techniques. P. Jiang, Q. Zhou, and X. Shao, *Surrogate Model-Based Engineering Design and Optimization*. Springer, 2020. A. Forrester, A. Sobester, and A. Keane, *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008.

17.1 Fitting Surrogate Models

A surrogate model $\hat{f}$ parameterized by θ is designed to mimic the true objective function f. The parameters θ can be adjusted to fit the model based on samples collected from f. An example surrogate model is shown in figure 17.1.

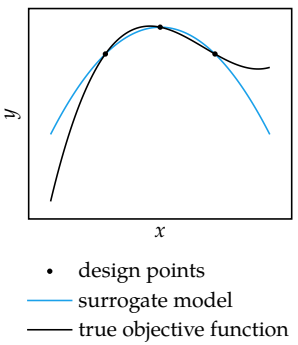


Figure 17.1. Surrogate models approximate the true objective function. The model is fitted to the evaluated design points but deviates farther away from them.

Suppose we have m design points

$$X = \left\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\right\} \tag{17.1}$$

and associated function evaluations

$$\mathbf{y} = \left\{y^{(1)}, y^{(2)}, \ldots, y^{(m)}\right\} \tag{17.2}$$

For a particular set of parameters, the model will predict

$$\hat{\mathbf{y}} = \left\{\hat{f}_\theta(\mathbf{x}^{(1)}), \hat{f}_\theta(\mathbf{x}^{(2)}), \ldots, \hat{f}_\theta(\mathbf{x}^{(m)})\right\} \tag{17.3}$$

Fitting a model to a set of points requires tuning the parameters to minimize the difference between the true evaluations and those predicted by the model, typically according to an L_p norm:[2]

$$\underset{\theta}{\text{minimize}} \quad \|\mathbf{y} - \hat{\mathbf{y}}\|_p \tag{17.4}$$

[2] It is common to use the L_2 norm. Minimizing this equation with an L_2 norm is equivalent to minimizing the mean squared error at those data points.

Equation (17.4) penalizes the deviation of the model only at the data points. There is no guarantee that the model will continue to fit well away from observed data, and model accuracy typically decreases the farther we go from the sampled points.

This form of model fitting is called *regression*. A large body of work exists for solving regression problems, and it is extensively studied in machine learning.[3] The rest of this chapter covers several popular surrogate models and algorithms for fitting surrogate models to data, and concludes with methods for choosing between types of models.

[3] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

17.2 Linear Models

A simple surrogate model is the *linear model*, which has the form[4]

[4] This equation may seem familiar. It is the equation for a hyperplane.

$$\hat{f} = w_0 + \mathbf{w}^\top \mathbf{x} \qquad \theta = \{w_0, \mathbf{w}\} \tag{17.5}$$

For an n-dimensional design space, the linear model has $n + 1$ parameters, and thus requires at least $n + 1$ samples to fit unambiguously.

Instead of having both $\mathbf{w}$ and w_0 as parameters, it is common to construct a single vector of parameters $\theta = [w_0, \mathbf{w}]$ and prepend 1 to the vector $\mathbf{x}$ to get

$$\hat{f} = \theta^\top \mathbf{x} \tag{17.6}$$

Finding an optimal θ requires solving a *linear regression* problem:

$$\underset{\theta}{\text{minimize}} \quad \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 \tag{17.7}$$

which is equivalent to solving

$$\underset{\theta}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{X}\theta\|_2^2 \tag{17.8}$$

where $\mathbf{X}$ is a *design matrix* formed from m data points

$$\mathbf{X} = \begin{bmatrix} (\mathbf{x}^{(1)})^\top \\ (\mathbf{x}^{(2)})^\top \\ \vdots \\ (\mathbf{x}^{(m)})^\top \end{bmatrix} \tag{17.9}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function design_matrix(X)
    n, m = length(X[1]), length(X)
    return [j==0 ? 1.0 : X[i][j] for i in 1:m, j in 0:n]
end
function linear_regression(X, y)
    θ = design_matrix(X) \ y
    return x -> θ⋅[1; x]
end
```

Algorithm 17.1. A method for constructing a design matrix from a list of design points `X` and a method for fitting a surrogate model using linear regression to a list of design points `X` and a vector of objective function values `y`.

Algorithm 17.1 implements methods for computing a design matrix and for solving a linear regression problem. Several cases for linear regression are shown in figure 17.2.

Linear regression has an analytic solution

$$\boldsymbol{\theta} = \mathbf{X}^{+}\mathbf{y} \tag{17.10}$$

where $\mathbf{X}^{+}$ is the Moore-Penrose *pseudoinverse* of $\mathbf{X}$, as covered in section 13.2.

17.3 Basis Functions

The linear model is a linear combination of the components of $\mathbf{x}$:

$$\hat{f}(\mathbf{x}) = \theta_1 x_1 + \cdots + \theta_n x_n = \sum_{i=1}^{n} \theta_i x_i = \boldsymbol{\theta}^\top \mathbf{x} \tag{17.11}$$

which is a specific example of a more general linear combination of *basis functions*

$$\hat{f}(\mathbf{x}) = \theta_1 b_1(\mathbf{x}) + \cdots + \theta_q b_q(\mathbf{x}) = \sum_{i=1}^{q} \theta_i b_i(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{b}(\mathbf{x}) \tag{17.12}$$

In the case of linear regression, the basis functions simply extract each component, $b_i(\mathbf{x}) = x_i$.

Any surrogate model represented as a linear combination of basis functions can be fit using regression:

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{B}\boldsymbol{\theta}\|_2^2 \tag{17.13}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

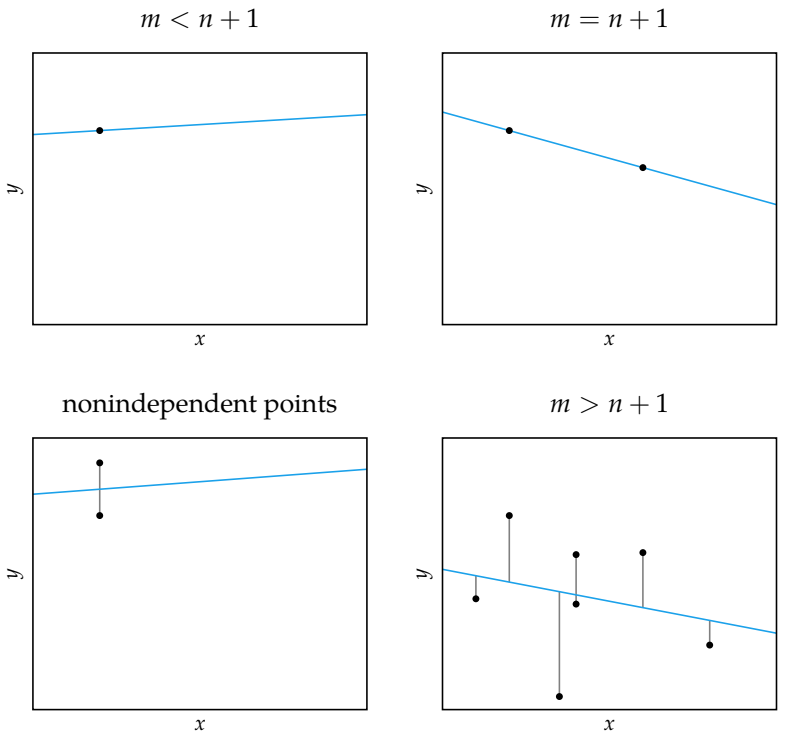


Figure 17.2. Models resulting from linear regression, which minimizes the square vertical distance of the model from each point. The pseudoinverse produces a unique solution for any nonempty point configuration.

The bottom-left subfigure shows the model obtained for two repeated points, in this case, $m = n + 1$. Because the two entries are repeated, the matrix $\mathbf{X}$ is singular. Although $\mathbf{X}$ does not have an inverse in this case, the pseudoinverse produces a unique solution that passes between the two points.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where $\mathbf{B}$ is the basis matrix formed from m data points:

$$\mathbf{B} = \begin{bmatrix} \mathbf{b}(\mathbf{x}^{(1)})^\top \\ \mathbf{b}(\mathbf{x}^{(2)})^\top \\ \vdots \\ \mathbf{b}(\mathbf{x}^{(m)})^\top \end{bmatrix} \tag{17.14}$$

The weighting parameters can be obtained using the pseudoinverse

$$\theta = \mathbf{B}^+ \mathbf{y} \tag{17.15}$$

Algorithm 17.2 implements this more general regression procedure.

```
using LinearAlgebra
function regression(X, y, bases)
    B = [b(x) for x in X, b in bases]
    θ = B \ y
    return x -> sum(θ[i] * bases[i](x) for i in eachindex(θ))
end
```

Algorithm 17.2. A method for fitting a surrogate model to a list of design points `X` and corresponding objective function values `y` using regression with basis functions contained in the `bases` array.

Linear models cannot capture nonlinear relations. There are a variety of other families of basis functions that can represent more expressive surrogate models. The remainder of this section discusses a few common families.

17.3.1 *Polynomial Basis Functions*

Polynomial basis functions consist of a product of design vector components, each raised to a power. Linear basis functions are a special case of polynomial basis functions.

From the Taylor series expansion[5] we know that any infinitely differentiable function can be closely approximated by a polynomial of sufficient degree. We can construct these bases using algorithm 17.3.

[5] Covered in appendix C.2.

In one dimension, a polynomial model of degree k has the form

$$\hat{f}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots + \theta_k x^k = \sum_{i=0}^{k} \theta_i x^i \tag{17.16}$$

Hence, we have a set of basis functions $b_i(x) = x^i$ for i ranging from 0 to k.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In two dimensions, a polynomial model of degree k has basis functions of the form

$$b_{ij}(\mathbf{x}) = x_1^i x_2^j \text{ for } i, j \in \{0, \ldots, k\}, \; i + j \leq k \tag{17.17}$$

Fitting a polynomial surrogate model is a regression problem, so a polynomial model is linear in higher dimensional space (figure 17.3). Any linear combination of basis functions can be viewed as linear regression in a higher dimensional space.

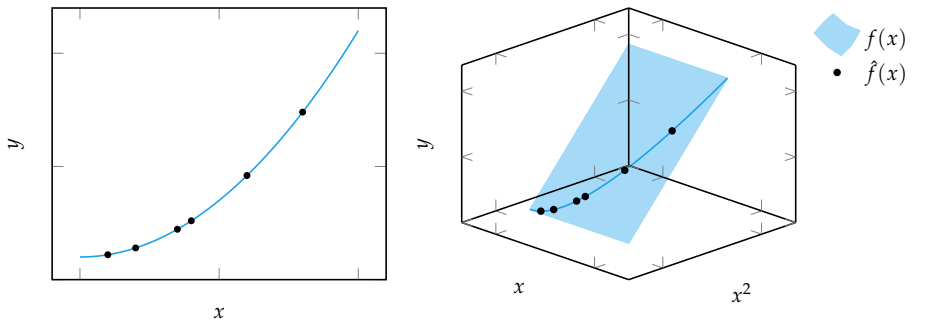


Figure 17.3. A polynomial model is linear in higher dimensions. The function exists in the plane formed from its bases, but it does not occupy the entire plane because the terms are not independent.

```
polynomial_bases_1d(i, k) = [x->x[i]^p for p in 0:k]
function polynomial_bases(n, k)
    bases = [polynomial_bases_1d(i, k) for i in 1 : n]
    terms = Function[]
    for ks in Iterators.product([0:k for i in 1:n]...)
        if sum(ks) ≤ k
            push!(terms,
                x->prod(b[j+1](x) for (j,b) in zip(ks,bases)))
        end
    end
    return terms
end
```

Algorithm 17.3. A method for constructing an array of polynomial basis functions up to a degree `k` for the `i`th component of a design point, and a method for constructing a list of n-dimensional polynomial bases up to degree `k`.

17.3.2 *Sinusoidal Basis Functions*

Any continuous function over a finite domain can be represented using an infinite set of *sinusoidal basis functions*.[6] A *Fourier series* can be constructed for any

[6] The Fourier series is also exact for functions defined over the entire real line if the function is periodic.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

integrable univariate function f on an interval $[a, b]$

$$f(x) = \frac{\theta_0}{2} + \sum_{i=1}^{\infty} \theta_i^{(\sin)} \sin\left(\frac{2\pi i x}{b-a}\right) + \theta_i^{(\cos)} \cos\left(\frac{2\pi i x}{b-a}\right) \tag{17.18}$$

where

$$\theta_0 = \frac{2}{b-a} \int_a^b f(x)\, dx \tag{17.19}$$

$$\theta_i^{(\sin)} = \frac{2}{b-a} \int_a^b f(x) \sin\left(\frac{2\pi i x}{b-a}\right) dx \tag{17.20}$$

$$\theta_i^{(\cos)} = \frac{2}{b-a} \int_a^b f(x) \cos\left(\frac{2\pi i x}{b-a}\right) dx \tag{17.21}$$

Just as the first few terms of a Taylor series are used in polynomial models, so too are the first few terms of the Fourier series used in sinusoidal models. The bases for a single component over the domain $x \in [a, b]$ are:

$$\begin{cases} b_0(x) & = 1/2 \\ b_i^{(\sin)}(x) & = \sin\left(\frac{2\pi i x}{b-a}\right) \\ b_i^{(\cos)}(x) & = \cos\left(\frac{2\pi i x}{b-a}\right) \end{cases} \tag{17.22}$$

We can combine the terms for multidimensional sinusoidal models in the same way we combine terms in polynomial models. Algorithm 17.4 can be used to construct sinusoidal basis functions. Several cases for sinusoidal regression are shown in figure 17.4.

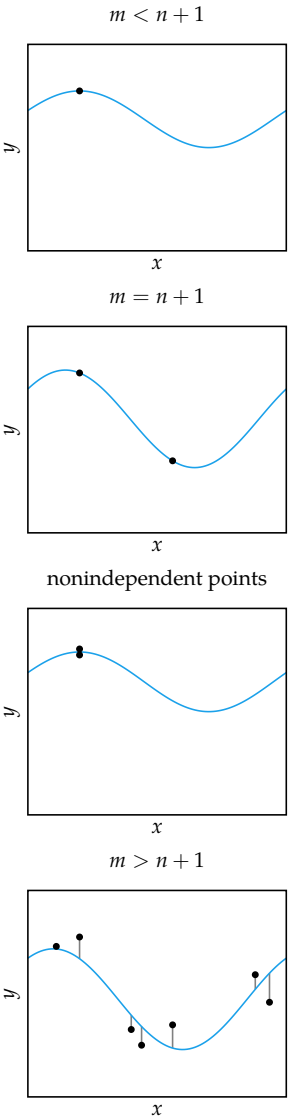


Figure 17.4. Fitting sinusoidal models to noisy points.

17.3.3 *Radial Basis Functions*

A *radial function*[7] ψ is a function that depends only on the distance of a point from some center point $\mathbf{c}$, such that it can be written $\psi(\mathbf{x}, \mathbf{c}) = \psi(\|\mathbf{x} - \mathbf{c}\|) = \psi(r)$. Radial functions are convenient basis functions because placing a radial function contributes a hill or valley to the function landscape. Some common radial basis functions are shown in figure 17.5.

Radial basis functions require specifying the center points. One approach when fitting radial basis functions to a set of data points is to use the data points as the centers. For a set of m points, one thus constructs m radial basis functions

$$b_i(\mathbf{x}) = \psi(\|\mathbf{x} - \mathbf{x}^{(i)}\|) \quad \text{for } i \text{ in } 1:m \tag{17.23}$$

[7] A comprehensive review is provided by M. D. Buhmann, *Radial Basis Functions*. Cambridge University Press, 2003.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function sinusoidal_bases_1d(j, k, a, b)
    T = b[j] - a[j]
    bases = Function[x->1/2]
    for i in 1 : k
        push!(bases, x->sin(2π*i*x[j]/T))
        push!(bases, x->cos(2π*i*x[j]/T))
    end
    return bases
end
function sinusoidal_bases(k, a, b)
    n = length(a)
    bases = [sinusoidal_bases_1d(i, k, a, b) for i in 1 : n]
    terms = Function[]
    for ks in Iterators.product([0:2k for i in 1:n]...)
        powers = [div(k+1,2) for k in ks]
        if sum(powers) ≤ k
            push!(terms,
                x->prod(b[j+1](x) for (j,b) in zip(ks,bases)))
        end
    end
    return terms
end
```

Algorithm 17.4. The method `sinusoidal_bases_1d` produces a list of basis functions up to degree `k` for the `j`th component of the design vector given lower bound `a` and upper bound `b`. The method `sinusoidal_bases` produces all basis function combinations up to degree `k` for lower-bound vector `a` and upper-bound vector `b`.

Algorithm 17.5 can be used to construct radial basis functions with known center points. Surrogate models with different radial basis functions are shown in figure 17.6.

```
radial_bases(ψ, C, p=2) = [x->ψ(norm(x - c, p)) for c in C]
```

Algorithm 17.5. A method for obtaining a list of basis functions given a radial basis function `ψ`, a list of centers `C`, and an L_p norm parameter `p`.

17.4 Fitting Noisy Objective Functions

Models fit using regression will pass as close as possible to every design point. When the objective function evaluations are noisy, complex models are likely to excessively contort themselves to pass through every point. However, smoother fits are often better predictors of the true underlying objective function.

The basis regression problem specified in equation (17.13) can be augmented to prefer smoother solutions. A *regularization term* is added in addition to the prediction error in order to give preference to solutions with lower weights. The resulting basis regression problem with L_2 *regularization*[8] is:

[8] Regression with L_2 regularization is also known as *ridge regression*. Other L_p-norms, covered in appendix C.4, can be used as well. Using the L_1 norm is known as the *lasso*, originally introduced in section 11.5.5. The lasso encourages sparse solutions with less influential component weights set to zero, which can be useful in identifying important basis functions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

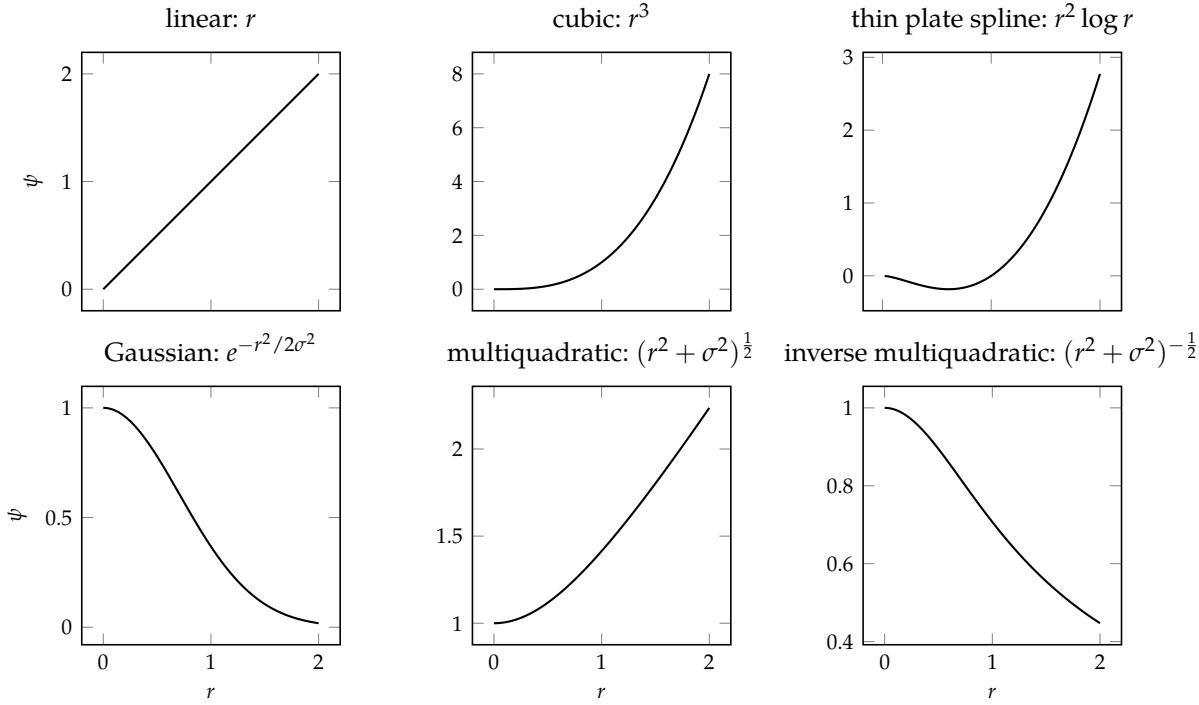


Figure 17.5. Several radial basis functions.

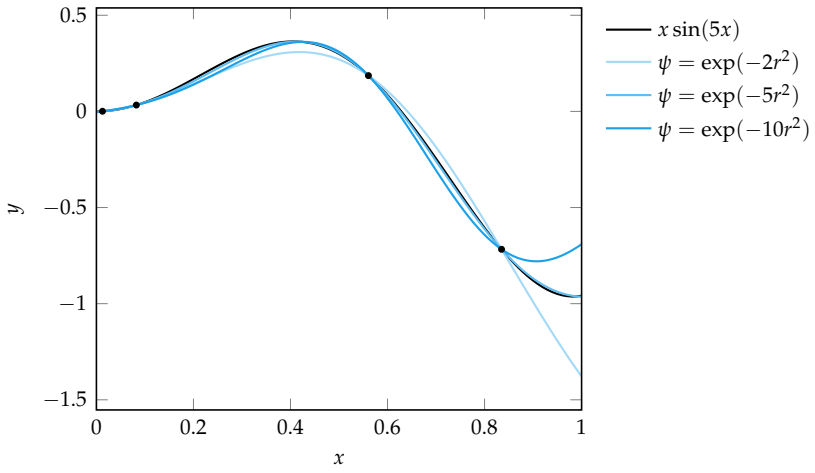


Figure 17.6. Several different Gaussian radial basis functions used to fit $x \sin(5x)$ based on four noise-free samples.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\underset{\theta}{\text{minimize}} \quad \|\mathbf{y} - \mathbf{B}\theta\|_2^2 + \lambda\|\theta\|_2^2 \tag{17.24}$$

where $\lambda \geq 0$ is a smoothing parameter, with $\lambda = 0$ resulting in no smoothing.[9]

The optimal parameter vector is given by:

$$\theta = \left(\mathbf{B}^\top\mathbf{B} + \lambda\mathbf{I}\right)^{-1}\mathbf{B}^\top\mathbf{y} \tag{17.25}$$

where $\mathbf{I}$ is the identity matrix. The matrix $\left(\mathbf{B}^\top\mathbf{B} + \lambda\mathbf{I}\right)$ is not always invertible if $\lambda = 0$. However, we can always produce an invertible matrix with a positive λ.

Algorithm 17.6 implements regression with L_2 regularization. Surrogate models with different radial basis functions fit to noisy samples are shown in figure 17.7.

[9] This optimization problem can be viewed as a multiobjective optimization problem (chapter 15) with the two objectives being minimize prediction error and minimize the sum of squared components of θ. We create a scalar objective function for this multiobjective problem using the weighted sum method from section 15.3.1.

```
function regression(X, y, bases, λ)
    B = [b(x) for x in X, b in bases]
    θ = (B'B + λ*I)\B'y
    return x -> sum(θ[i] * bases[i](x) for i in eachindex(θ))
end
```

Algorithm 17.6. A method for regression in the presence of noise, where `λ` is a smoothing term. It returns a surrogate model fitted to a list of design points `X` and corresponding objective function values `y` using regression with basis functions `bases`.

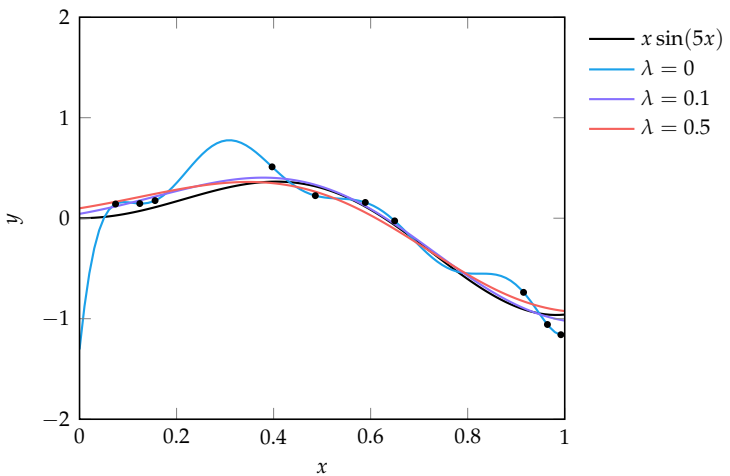


Figure 17.7. Several different smoothing parameters λ used to fit $x\sin(5x)$ with zero mean, standard deviation 0.1 error based on ten noisy samples and radial basis function $\psi = \exp(-5r^2)$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

17.5 Model Selection

So far, we have discussed how to fit a particular model to data. This section explains how to select which model to use. We generally want to minimize *generalization error*, which is a measure of the error of the model on the full design space, including points that may not be included in the data used to train the model. One way to measure generalization error is to use the expected squared error of its predictions:

$$\epsilon_{\text{gen}} = \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[\left(f(\mathbf{x}) - \hat{f}(\mathbf{x})\right)^2\right] \tag{17.26}$$

Of course, we cannot calculate this generalization error exactly because it requires knowing the function we are trying to approximate. It may be tempting to estimate the generalization error of a model from the *training error*. One way to measure training error is to use the *mean squared error* (MSE) of the model evaluated on the m samples used for training:

$$\epsilon_{\text{train}} = \frac{1}{m}\sum_{i=1}^{m}\left(f(\mathbf{x}^{(i)}) - \hat{f}(\mathbf{x}^{(i)})\right)^2 \tag{17.27}$$

However, performing well on the training data does not necessarily correspond to low generalization error. Complex models may reduce the error on the training set, but they may not provide good predictions at other points in the design space as illustrated in example 17.1.[10]

This section discusses several methods for estimating generalization error. These methods train and test on subsets of the data with the help of algorithm 17.7. Although we train on subsets of the data when estimating the generalization error, once we have decided which model to use, we can train on the full dataset.

[10] A major theme in machine learning is balancing model complexity to avoid *overfitting* the training data. K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

```
struct TrainTest
    train # training indices
    test  # testing indices
end
function train_and_validate(X, y, tt, fit, metric)
    model = fit(X[tt.train], y[tt.train])
    return metric(model, X[tt.test], y[tt.test])
end
```

Algorithm 17.7. A method for training a model and then validating it on a metric. Here, `train` and `test` are lists of indices into the training data, `X` is a list of design points, `y` is the vector of corresponding function evaluations, `tt` is a train-test partition, `fit` is a model fitting function, and `metric` evaluates a model on test data.

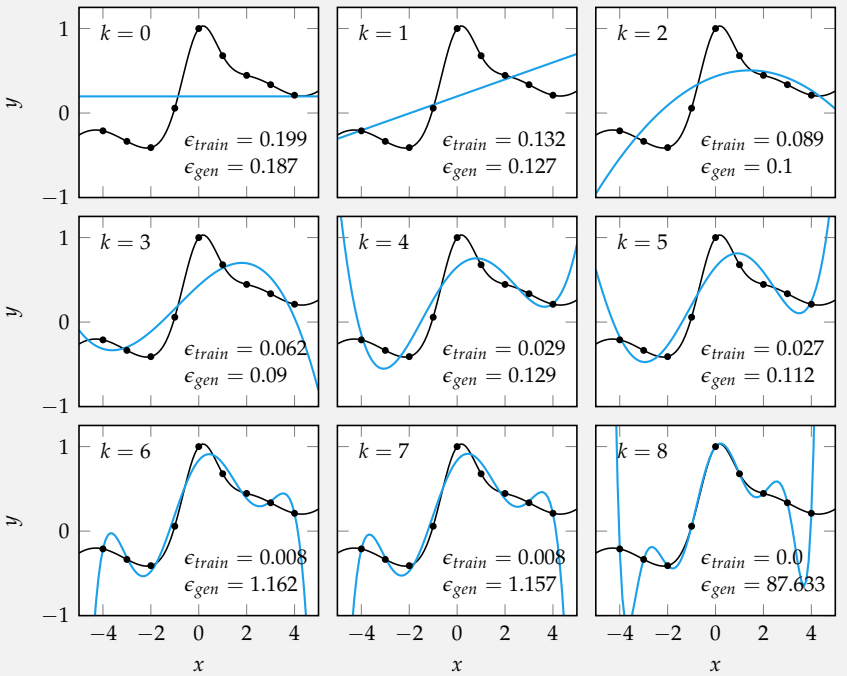
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 17.1. A comparison of training and generalization error as the degree of a polynomial surrogate model is varied.

Consider fitting polynomials of varying degrees to evaluations of the objective function

$$f(x) = x/10 + \sin(x)/4 + \exp\left(-x^2\right)$$

Below we plot polynomial surrogate models of varying degrees using the same nine evaluations evenly spaced over $[-4, 4]$. The training and generalization error are shown as well, where generalization is calculated over $[-5, 5]$.



The plot shows that the generalization error is high for both very low and high values of k, and that training error decreases as we increase the polynomial degree. The high-degree polynomials are particularly poor predictors for designs outside $[-4, 4]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

17.5.1 Holdout



Figure 17.8. The holdout method partitions the data into train and test sets.

$$\text{train}(\bullet) \longrightarrow \text{test}(\hat{f}, \bullet) \longrightarrow \text{generalization error estimate}$$

A simple approach to estimating the generalization error is the *holdout method*, which partitions the available data into a *test set* $\mathcal{D}_h$ with h samples and a *training set* $\mathcal{D}_t$ consisting of all remaining $m - h$ samples as shown in figure 17.8. The training set is used to fit model parameters. The held out test set is not used during model fitting, and can thus be used to estimate the generalization error. Different split ratios are used, typically ranging from 50% train, 50% test to 90% train, 10% test, depending on the size and nature of the dataset. Using too few samples for training can result in poor fits (figure 17.9), whereas using too many will result in poor generalization estimates.

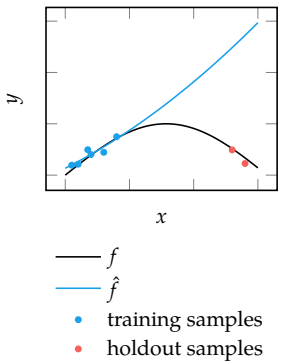


Figure 17.9. Poor train-test splits can result in poor model performance.

The holdout error for a model $\hat{f}$ fit to the training set is

$$\epsilon_{\text{holdout}} = \frac{1}{h} \sum_{(\mathbf{x},y)\in\mathcal{D}_h} \left(y - \hat{f}(\mathbf{x})\right)^2 \tag{17.28}$$

```
function holdout_partition(m, h=div(m,2))
    p = randperm(m)
    train = p[(h+1):m]
    holdout = p[1:h]
    return TrainTest(train, holdout)
end
```

Algorithm 17.8. A method for randomly partitioning `m` data samples into training and holdout sets, where `h` samples are assigned to the holdout set.

Even if the partition ratio is fixed, the holdout error will depend on the particular train-test partition chosen. Choosing a partition at random (algorithm 17.8) will only give a point estimate. In *random subsampling* (algorithm 17.9), we apply the holdout method multiple times with randomly selected train-test partitions. The estimated generalization error is the mean over all runs.[11] Because the validation sets are chosen randomly, this method does not guarantee that we validate on all of the data points.

[11] The standard deviation over all runs can be used to estimate the standard deviation of the estimated generalization error.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function random_subsampling(X, y, fit, metric;
    h=div(length(X),2), k_max=10)
    m = length(X)
    mean(train_and_validate(X, y, holdout_partition(m, h),
          fit, metric) for k in 1 : k_max)
end
```

Algorithm 17.9. The random subsampling method used to obtain a mean estimate for model generalization error using `k_max` runs of the holdout method.

17.5.2 *Cross-Validation*

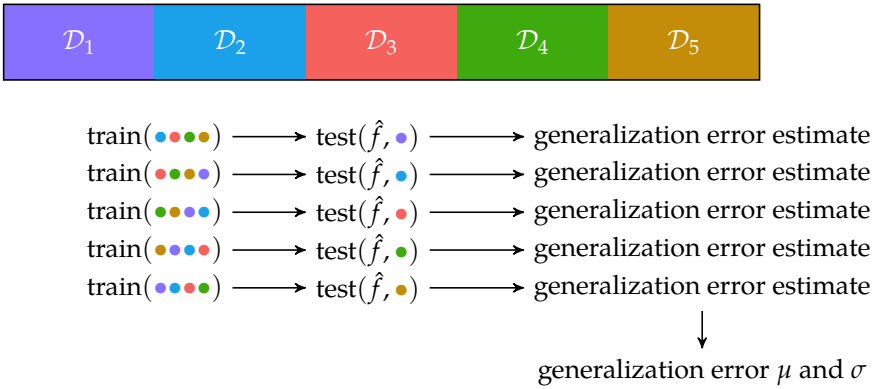


Figure 17.10. Cross-validation partitions the data into equally sized sets. Each set is the holdout set once. Here we show five-fold cross-validation.

Using a train-test partition can be wasteful because our model tuning can take advantage only of a segment of our data. Better results are often obtained by using *k-fold cross-validation*.[12] Here, the original dataset $\mathcal{D}$ is randomly partitioned into k sets $\mathcal{D}_1, \ldots, \mathcal{D}_k$ of equal, or approximately equal, size, as shown in figure 17.10 and implemented in algorithm 17.10. We then train k models, one on each subset of $k - 1$ sets, and we use the withheld set to estimate the generalization error. The cross-validation estimate of generalization error is the mean generalization error over all folds:[13]

$$\epsilon_{\text{cross-validation}} = \frac{1}{k}\sum_{i=1}^{k} \epsilon^{(i)}_{\text{cross-validation}} \tag{17.29}$$

$$\epsilon^{(i)}_{\text{cross-validation}} = \frac{1}{|\mathcal{D}^{(i)}_{\text{test}}|} \sum_{(\mathbf{x},y)\in\mathcal{D}^{(i)}_{\text{test}}} \left(y - \hat{f}^{(i)}(\mathbf{x})\right)^2 \tag{17.30}$$

where $\epsilon^{(i)}_{\text{cross-validation}}$ and $\mathcal{D}^{(i)}_{\text{test}}$ are the cross-validation estimate and the withheld test set, respectively, for the ith fold.

[12] Also known as *rotation estimation*.

[13] As with random subsampling, an estimate of variance can be obtained from the standard deviation over folds.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function k_fold_cross_validation_sets(m, k)
    perm = randperm(m)
    sets = TrainTest[]
    for i = 1:k
        validate = perm[i:k:m];
        train = perm[setdiff(1:m, i:k:m)]
        push!(sets, TrainTest(train, validate))
    end
    return sets
end
function cross_validation_estimate(X, y, sets, fit, metric)
    mean(train_and_validate(X, y, tt, fit, metric) for tt in sets)
end
```

Algorithm 17.10. The method `k_fold_cross_validation_sets` constructs the sets needed for `k`-fold cross validation on `m` samples, with `k ≤ m`. The method `cross_validation_estimate` computes the mean of the generalization error estimate by training and validating on the list of train-validate sets contained in `sets`. The other variables are the list of design points `X`, the corresponding objective function values `y`, a function `fit` that trains a surrogate model, and a function `metric` that evaluates a model on a dataset.

Cross-validation also depends on the particular data partition. An exception is *leave-one-out cross-validation* with $k = m$, which has a deterministic partition. It trains on as much data as possible, but it requires training m models.[14] Averaging over all $\binom{m}{m/k}$ possible partitions, known as *complete cross-validation*, is often too expensive. While one can average multiple cross-validation runs, it is more common to average the models from a single cross-validation partition.

Cross-validation is demonstrated in example 17.2.

[14] M. Stone, "Cross-Validatory Choice and Assessment of Statistical Predictions," *Journal of the Royal Statistical Society*, vol. 36, no. 2, pp. 111–147, 1974.

17.5.3 The Bootstrap

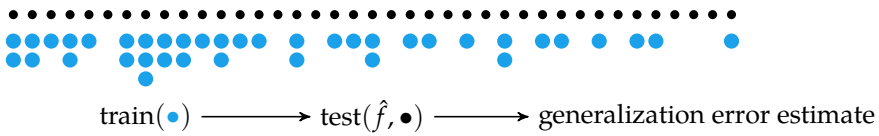


Figure 17.11. A single bootstrap sample consists of m indices into the dataset sampled with replacement. A bootstrap sample is used to train a model, which is evaluated on the full dataset to obtain an estimate of the generalization error.

The *bootstrap method*[15] uses multiple *bootstrap samples*, which consist of m indices into a dataset of size m independently chosen uniformly at random. The indices are chosen with replacement, so some indices may be chosen multiple times and some indices may not be chosen at all as shown in figure 17.11. The bootstrap sample is used to fit a model that is then evaluated on the original training set. A method for obtaining bootstrap samples is given in algorithm 17.11.

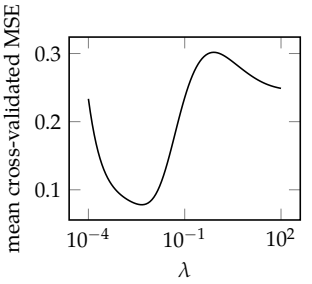
[15] B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *The Annals of Statistics*, vol. 7, pp. 1–26, 1979.

```
bootstrap_sets(m, b) = [TrainTest(rand(1:m, m), 1:m) for i in 1:b]
```

Algorithm 17.11. A method for obtaining `b` bootstrap samples, each for a dataset of size `m`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 17.2. Cross-validation used to fit a hyperparameter.



Suppose we want to fit a noisy objective function using radial basis functions with the noise hyperparameter λ (section 17.4). We can use cross-validation to determine λ. We are given ten samples from our noisy objective function. In practice, the objective function will be unknown, but this example uses

$$f(x) = \sin(2x)\cos(10x) + \epsilon/10$$

where $x \in [0, 1]$ and ϵ is random noise with zero mean and unit variance, $\epsilon \sim \mathcal{N}(0, 1)$.

```
Random.seed!(0)
f = x->sin(2x)*cos(10x)
X = rand(10)
y = f.(X) + randn(length(X))/10
```

We will use three folds assigned randomly:

```
sets = k_fold_cross_validation_sets(length(X), 3)
```

Next, we implement our metric. We use the mean squared error:

```
metric = (f, X, y)->begin
    m = length(X)
    return sum((f(X[i]) - y[i])^2 for i in 1:m)/m
end
```

We now loop through different values of λ and fit different radial basis functions. We will use the Gaussian radial basis. Cross-validation is used to obtain the MSE for each value:

```
λs = 10 .^ range(-4, stop=2, length=101)
es = []
basis = r->exp(-5r^2)
for λ in λs
    fit = (X, y)->regression(X, y, radial_bases(basis, X), λ)
    push!(es,
          cross_validation_estimate(X, y, sets, fit, metric))
end
```

The resulting curve has a minimum at $\lambda \approx 0.07$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If b bootstrap samples are made, then the bootstrap estimate of the generalization error is the mean of the corresponding generalization error estimates $\epsilon_{\text{test}}^{(1)}, \ldots, \epsilon_{\text{test}}^{(b)}$:

$$\epsilon_{\text{boot}} = \frac{1}{b}\sum_{i=1}^{b} \epsilon_{\text{test}}^{(i)} \tag{17.31}$$

$$= \frac{1}{m}\sum_{j=1}^{m} \frac{1}{b}\sum_{i=1}^{b} \left(y^{(j)} - \hat{f}^{(i)}(\mathbf{x}^{(j)})\right)^2 \tag{17.32}$$

where $\hat{f}^{(i)}$ is the model fit to the ith bootstrap sample. The bootstrap method is implemented in algorithm 17.12.

```
function bootstrap_estimate(X, y, sets, fit, metric)
    mean(train_and_validate(X, y, tt, fit, metric) for tt in sets)
end
```

Algorithm 17.12. A method for computing the bootstrap generalization error estimate by training and validating on the list of train-validate sets contained in `sets`. The other variables are the list of design points `X`, the corresponding objective function values `y`, a function `fit` that trains a surrogate model, and a function `metric` that evaluates a model on a dataset.

The bootstrap error in equation (17.31) tests models on data points to which they were fit. The *leave-one-out bootstrap estimate* removes this source of bias by only evaluating fitted models on withheld data:

$$\epsilon_{\text{leave-one-out-boot}} = \frac{1}{m}\sum_{j=1}^{m} \frac{1}{c_{-j}} \sum_{i=1}^{b} \begin{cases} \left(y^{(j)} - \hat{f}^{(i)}(\mathbf{x}^{(j)})\right)^2 & \text{if } j\text{th index was not in the } i\text{th bootstrap sample} \\ 0 & \text{otherwise} \end{cases} \tag{17.33}$$

where c_{-j} is the number of bootstrap samples that do not contain index j. The leave-one-out bootstrap method is implemented in algorithm 17.13.

The probability of a particular index not being in a bootstrap sample is:

$$\left(1 - \frac{1}{m}\right)^m \approx e^{-1} \approx 0.368 \tag{17.34}$$

so a bootstrap sample is expected to have on average $0.632m$ distinct indices from the original dataset.

Unfortunately, the leave-one-out bootstrap estimate introduces a new bias due to the varying test set sizes. The 0.632 *bootstrap estimate*[16] (algorithm 17.14) alleviates this bias:

$$\epsilon_{0.632\text{-boot}} = 0.632\epsilon_{\text{leave-one-out-boot}} + 0.368\epsilon_{\text{boot}} \tag{17.35}$$

Several generalization estimation methods are compared in example 17.3.

[16] The 0.632 bootstrap estimate was introduced in B. Efron, "Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation," *Journal of the American Statistical Association*, vol. 78, no. 382, pp. 316–331, 1983. A variant, the 0.632+ bootstrap estimate, was introduced in B. Efron and R. Tibshirani, "Improvements on Cross-Validation: The .632+ Bootstrap Method," *Journal of the American Statistical Association*, vol. 92, no. 438, pp. 548–560, 1997.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function leave_one_out_bootstrap_estimate(X, y, sets, fit, metric)
    m, b = length(X), length(sets)
    ε = 0.0
    models = [fit(X[tt.train], y[tt.train]) for tt in sets]
    for j in 1 : m
        c = 0
        δ = 0.0
        for i in 1 : b
            if j ∉ sets[i].train
                c += 1
                δ += metric(models[i], [X[j]], [y[j]])
            end
        end
        ε += δ/c
    end
    return ε/m
end
```

Algorithm 17.13. A method for computing the leave-one-out bootstrap generalization error estimate using the train-validate sets `sets`. The other variables are the list of design points `X`, the corresponding objective function values `y`, a function `fit` that trains a surrogate model, and a function `metric` that evaluates a model on a dataset.

```
function bootstrap_632_estimate(X, y, sets, fit, metric)
    models = [fit(X[tt.train], y[tt.train]) for tt in sets]
    ϵ_loob = leave_one_out_bootstrap_estimate(X,y,sets,fit,metric)
    ϵ_boot = bootstrap_estimate(X,y,sets,fit,metric)
    return 0.632ϵ_loob + 0.368ϵ_boot
end
```

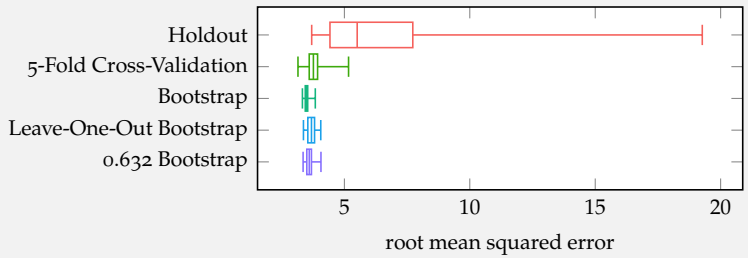
Algorithm 17.14. A method for obtaining the 0.632 bootstrap estimate for data points `X`, objective function values `y`, train-validation sets `sets`, fitting function `fit`, and metric function `metric`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 17.3. A comparison of generalization error estimation methods. The vertical lines in the *box and whisker* plots indicate the minimum, maximum, first and third quartiles, and median of every generalization error estimation method among 100 trials.

Consider ten evenly spread samples of $f(x) = x^2 + \epsilon/2$ over $x \in [-3, 3]$, where ϵ is zero-mean, unit-variance Gaussian noise. We would like to test several different generalization error estimation methods when fitting a linear model to this data. Our metric is the *root mean squared error*, which is the square root of the mean squared error.

The methods used are the holdout method with eight holdout samples, five-fold cross-validation, and the bootstrap methods each with fifty bootstrap samples. Each method was fitted 100 times and the resulting statistics are shown below.



17.6 Multifidelity Surrogate Models

We use the word *fidelity* to describe the level of detail or accuracy of a model or a dataset. For some applications, high-fidelity data is expensive and time-consuming to obtain in comparison to low-fidelity data, which can be much less expensive to obtain, but also less accurate. In the context of aircraft design, high-fidelity data might be obtained from a full-scale flight test, whereas low-fidelity data might be obtained from a fluid dynamics simulator. This section discusses how to use both low-fidelity and high-fidelity data to produce a *multifidelity surrogate model*.

In this section, we assume we have two datasets, a low-fidelity dataset X_ℓ and a high-fidelity dataset X_h. Typically, the size of X_ℓ is much larger than the size of X_h. Associated with these datasets are different evaluation functions, $f_\ell(\mathbf{x})$ and $f_h(\mathbf{x})$. We assume that $f_\ell(\mathbf{x})$ is a noisy approximation of $f_h(\mathbf{x})$. We want to use both datasets to produce a surrogate model $\hat{f}_h(\mathbf{x})$ that is a good approximation of $f_h(\mathbf{x})$.

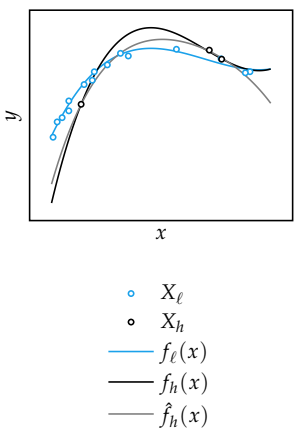


Figure 17.12. A high-fidelity surrogate model $\hat{f}_h(x)$ produced using many noisy low-fidelity samples and a small number of noisy high-fidelity samples.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

There are many methods for constructing $\hat{f}_h(\mathbf{x})$ from X_ℓ and X_h. One approach is to train a low-fidelity surrogate model $\hat{f}_\ell(\mathbf{x})$ using X_ℓ and then use X_h to find a transformation that makes $\hat{f}_\ell(\mathbf{x})$ match $f_h(\mathbf{x})$ as closely as possible. This transformation can take different forms, such as an affine transformation defined by scaling and offset parameters. An affine transformation would lead to a multifidelity model with the form:

$$\hat{f}_h(\mathbf{x}) = a_0 + a_1\hat{f}_\ell(\mathbf{x}) \tag{17.36}$$

where a_0 and a_1 are free parameters in the model. This transformation is just another surrogate model, and the parameters can be fit using the high-fidelity data just as in section 17.2.

Of course, an affine model might be inadequate to capture the differences between $f_\ell(\mathbf{x})$ and $f_h(\mathbf{x})$. One way to address this is to construct a discrepancy model $\hat{\delta}(\mathbf{x})$ that captures the differences between $\hat{f}_h(\mathbf{x})$ and $f_h(\mathbf{x})$. This discrepancy model could be constructed using any of the methods discussed earlier in this chapter.

There are many extensions to the multifidelity surrogate model framework:[17]

- Multifidelity surrogate models where f_ℓ and f_h do not have identical design vectors. In such cases, data from the low-fidelity model can be projected into the high-fidelity space.
- Cases where f_ℓ and f_h do not measure directly comparable outputs. For example, test kitchens may be able to predict or measure objective qualities of a new dish like sweetness and salinity (f_ℓ), but the true higher-cost objective is based on how much their customers enjoy the resulting dish (f_h). A common strategy is to build a mapping between outputs using samples of both f_ℓ and f_h from the same design vectors.
- Nonhierarchical datasets where f_h is not necessarily higher-fidelity than f_ℓ, and multiple models of similar, or unknown, fidelity must be combined. One approach is to construct a surrogate model for each data source, and then to combine predictions from each model.

[17] For a broad overview, see Q. Zhou, M. Zhao, J. Hu, and M. Ma, *Multi-Fidelity Surrogates: Modeling, Optimization and Applications*. Springer, 2023.

17.7 Summary

- Surrogate models are function approximations that can be optimized instead of the true, potentially expensive objective function.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- Many surrogate models can be represented using a linear combination of basis functions.
- Model selection involves a bias-variance trade-off between models with low complexity that cannot capture important trends and models with high complexity that overfit to noise.
- Generalization error can be estimated using techniques such as holdout, k-fold cross-validation, and the bootstrap.
- Multifidelity surrogate models can be constructed to use multiple data sources with varying fidelity.

17.8 Exercises

Exercise 17.1. Derive an expression satisfied by the optimum of the regression problem equation (17.8) by setting the gradient to zero. Do not invert any matrices. The resulting relation is called the *normal equation*.

Solution: The linear regression objective function is

$$\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$$

We take the gradient and set it to zero:

$$\nabla(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) = -2\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) = \mathbf{0}$$

which yields the normal equation

$$\mathbf{X}^\top\mathbf{X}\boldsymbol{\theta} = \mathbf{X}^\top\mathbf{y}$$

Exercise 17.2. When would we use a more descriptive model, for example, with polynomial features, versus a simpler model like linear regression?

Solution: As a general rule, more descriptive models should be used when more data are available. If only few samples are available, such models are prone to overfitting, and a simpler model (with fewer degrees of freedom) should be used.

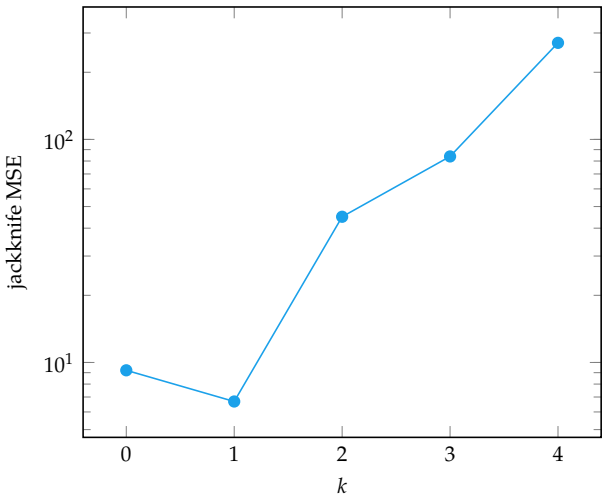
Exercise 17.3. A linear regression problem of the form in equation (17.8) is not always solved analytically, and optimization techniques are used instead. Why is this the case?

Solution: The model at hand may have a very large number of parameters. In such a case, the resulting linear system will be too large and will require memory that grows quadratically with the parameter space. Iterative procedures like stochastic gradient descent require memory linear in the size of the parameter space and are sometimes the only viable solution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 17.4. Suppose we evaluate our objective function at four points: 1, 2, 3, and 4, and we get back 0, 5, 4, and 6. We want to fit a polynomial model $f(x) = \sum_{i=0}^{k} \theta_i x^i$. Compute the leave-one-out cross-validation estimate of the mean squared error as k varies between 0 and 4. According to this metric, what is the best value for k, and what are the best values for the elements of θ?

Solution: The leave-one-out cross-validation estimate is obtained by running k-fold cross-validation with k equal to the number of samples in X. This means we must run four-fold cross-validation for each polynomial degree.



The lowest mean squared error is obtained for a linear model, $k = 1$. We fit a new linear model on the complete dataset to obtain our parameters:

```
X = [[1],[2],[3],[4]]
y = [0,5,4,6]
bases = polynomial_bases(1, 1)
B = [b(x) for x in X, b in bases]
θ = B \ y
@show θ
```

```
θ = [-0.500000000000001, 1.7000000000000002]
```

Exercise 17.5. Suppose you have an aircraft collision avoidance system that you wish to optimize to minimize the frequency of near mid-air collisions, and you have a simulator that can simulate your collision avoidance system on generated aircraft encounters. You can run the simulator many times to get an accurate assessment of the collision avoidance system's performance. You can also run the simulator fewer times on more dangerous scenarios to get a quicker, albeit less accurate, assessment.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Suppose your high-fidelity simulator evaluations execute a million flight hours worth of simulations, of which only 1% are safety-critical.[18] Suppose also that your low-fidelity simulator runs only a thousand flight hours worth of simulations, but they are heavily biased such that 100% of them are safety-critical.

Notionally describe $\mathbf{x}$, f_ℓ, f_h, X_ℓ, X_h, and our optimization objective. If you were to construct a multifidelity surrogate model using equation (17.36), what would you expect a_0 and a_1 to be?

[18] Scenarios that are not safety-critical are unlikely to result in a near mid-air collision.

Solution: Here we tune the parameters of an aircraft collision avoidance system $\mathbf{x}$ in order to minimize the frequency of near mid-air collisions. The entries of $\mathbf{x}$ determine the behavior of the collision avoidance system.

We have access to a simulator `sim`, which typically takes a scenario description $\mathbf{s}$ containing initial conditions and descriptions of scenario-related events, and then simulates the behavior of two or more aircraft and the parameterized aircraft collision avoidance system to produce a trajectory, $\texttt{sim}(\mathbf{x}, \mathbf{s}) \to \tau$. We quantify the performance of the collision avoidance system by evaluating whether the resulting trajectory includes a near mid-air collision, $\texttt{has\_nmac}(\tau)$. Our optimization objective is to minimize the frequency of near mid-air collisions under some distribution of scenarios S:

$$\underset{\mathbf{x}}{\text{minimize}} \; \mathop{\mathbb{E}}_{\mathbf{s} \in S}[\texttt{has\_nmac}(\texttt{sim}(\mathbf{x}, \mathbf{s}))]$$

The functions f_ℓ and f_h both notionally have the form:

```
function objective(x, S, m)
    n_nmacs = 0
    for i in 1:m
        s = rand(S)
        τ = simulate(x, s)
        n_nmacs += has_nmac(τ)
    end
    return n_nmacs / m
end
```

They differ in both the distribution of scenarios S and the number of samples m, with f_h simulating 1,000 times more scenarios than f_ℓ assuming scenario durations are roughly uniform in length. The scenario distribution for f_ℓ is heavily biased toward safety-critical scenarios.

The low-fidelity function f_ℓ is thus a thousand times cheaper to evaluate than f_h. We can afford to sweep over the design space using f_ℓ, producing a comparably large X_ℓ. In comparison, X_h will have far fewer evaluations. Both of these sets will consist of (design, estimated collision frequency) tuples, though estimated collision frequencies using f_ℓ will be much larger due to the scenario biasing.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In constructing our multifidelity surrogate model, we expect to have to scale down the frequency of near mid-air collisions obtained from the low-fidelity simulator in order to estimate the true safety risk. In this case, we do not need a bias term, so a_0 would likely be close to zero. We would expect a_1 to be roughly 0.01 to account for the adjustment in criticality.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

18 Probabilistic Surrogate Models

The previous chapter discussed how to construct surrogate models from evaluated design points. When using surrogate models for the purpose of optimization, it is often useful to quantify our confidence in the predictions of these models. One way to quantify our confidence is by taking a probabilistic approach to surrogate modeling. A common probabilistic surrogate model is the *Gaussian process*, which represents a probability distribution over functions. This chapter will explain how to use Gaussian processes to infer a distribution over the values of different design points given the values of previously evaluated design points. We will discuss how to incorporate gradient information as well as noisy measurements of the objective function. Since the predictions made by a Gaussian process are governed by a set of parameters, we will discuss how to infer these parameters directly from data.

18.1 Gaussian Distribution

Before introducing Gaussian processes, we will first review some relevant properties of the multivariate *Gaussian distribution*, often also referred to as the multivariate *normal distribution*.[1] An n-dimensional Gaussian distribution is parameterized by its mean $\boldsymbol{\mu}$ and its covariance matrix $\boldsymbol{\Sigma}$. The probability density at $\mathbf{x}$ is

[1] The univariate Gaussian distribution is discussed in appendix C.9.

$$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-n/2}|\boldsymbol{\Sigma}|^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{18.1}$$

Figure 18.1 shows contour plots of the density functions with different covariance matrices. Covariance matrices are always positive semidefinite.

A value sampled from a Gaussian is written

$$\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{18.2}$$

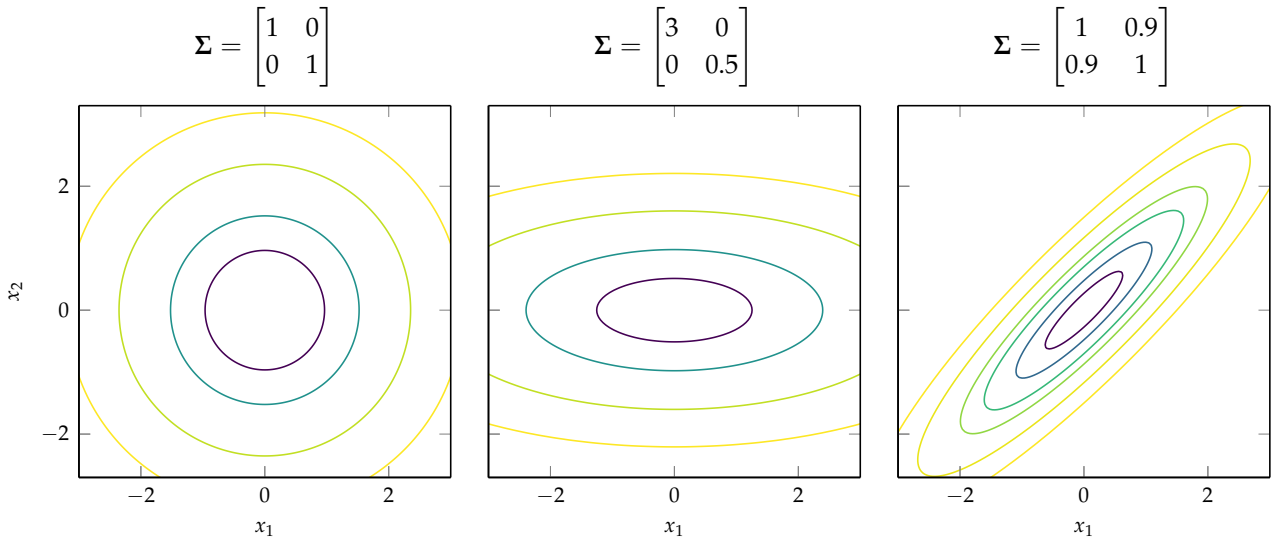


Figure 18.1. Multivariate Gaussians with different covariance matrices.

Two jointly Gaussian random variables $\mathbf{a}$ and $\mathbf{b}$ can be written

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_\mathbf{a} \\ \boldsymbol{\mu}_\mathbf{b} \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{C} \\ \mathbf{C}^\top & \mathbf{B} \end{bmatrix} \right) \tag{18.3}$$

The *marginal distribution*[2] for a vector of random variables is given by its corresponding mean and covariance

$$\mathbf{a} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{a}, \mathbf{A}) \qquad \mathbf{b} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{b}, \mathbf{B}) \tag{18.4}$$

The *conditional distribution* for a multivariate Gaussian also has a convenient closed-form solution:

$$\mathbf{a} \mid \mathbf{b} \sim \mathcal{N}\left(\boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}}, \boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}}\right) \tag{18.5}$$

$$\boldsymbol{\mu}_{\mathbf{a}|\mathbf{b}} = \boldsymbol{\mu}_\mathbf{a} + \mathbf{C}\mathbf{B}^{-1}(\mathbf{b} - \boldsymbol{\mu}_\mathbf{b}) \tag{18.6}$$

$$\boldsymbol{\Sigma}_{\mathbf{a}|\mathbf{b}} = \mathbf{A} - \mathbf{C}\mathbf{B}^{-1}\mathbf{C}^\top \tag{18.7}$$

Example 18.1 illustrates how to extract the marginal and conditional distributions from a multivariate Gaussian.

[2] The marginal distribution is the distribution of a subset of the variables when the rest are integrated, or marginalized, out. For a distribution over two variables a and b the marginal distribution over a is:

$$p(a) = \int p(a, b)\, db$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 18.1. Marginal and conditional distributions for a multivariate Gaussian.

We have a joint Gaussian distribution over two variables x_1 and x_2:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \right)$$

The marginal distribution for x_1 is $\mathcal{N}(0, 3)$, and the marginal distribution for x_2 is $\mathcal{N}(1, 2)$.

The conditional distribution for x_1 given $x_2 = 2$ is

$$\begin{aligned} \boldsymbol{\mu}_{x_1 \mid x_2 = 2} &= 0 + 1 \cdot 2^{-1} \cdot (2 - 1) = 0.5 \\ \boldsymbol{\Sigma}_{x_1 \mid x_2 = 2} &= 3 - 1 \cdot 2^{-1} \cdot 1 = 2.5 \\ x_1 \mid (x_2 = 2) &\sim \mathcal{N}(0.5, 2.5) \end{aligned}$$

18.2 Gaussian Processes

In the previous chapter, we approximated the objective function f using a surrogate model function $\hat{f}$ fitted to previously evaluated design points. A special type of surrogate model known as a *Gaussian process* allows us not only to predict f but also to quantify our uncertainty in that prediction using a probability distribution.[3]

A Gaussian process is a distribution over functions. For any finite set of points $X = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(|X|)}\}$, the associated function evaluations $\left\{y_1, \ldots, y_{|X|}\right\}$ are distributed according to:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{|X|} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathbf{x}^{(1)}) \\ \vdots \\ m(\mathbf{x}^{(|X|)}) \end{bmatrix}, \begin{bmatrix} k(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \cdots & k(\mathbf{x}^{(1)}, \mathbf{x}^{(|X|)}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}^{(|X|)}, \mathbf{x}^{(1)}) & \cdots & k(\mathbf{x}^{(|X|)}, \mathbf{x}^{(|X|)}) \end{bmatrix} \right) \tag{18.8}$$

where $m(\mathbf{x})$ is a *mean function* and $k(\mathbf{x}, \mathbf{x}')$ is the *covariance function*, or *kernel*.[4] The mean function can represent prior knowledge about the function. The kernel controls the smoothness of the functions. Methods for constructing the mean vector and covariance matrix using mean and covariance functions are given in algorithm 18.1.

[3] A more extensive introduction to Gaussian processes is provided by C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006. The `GaussianProcesses.jl` package provides some implementation. J. Fairbrother, C. Nemeth, M. Rischard, J. Brea, and T. Pinder, "GaussianProcesses.jl: A Nonparametric Bayes Package for the Julia Language," *Journal of Statistical Software*, vol. 102, no. 1, 2022.

[4] The mean function produces the expectation:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})]$$

and the covariance function produces the covariance:

$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}\left[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))\right]$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
μ(X, m) = [m(x) for x in X]
K(X, X′, k) = [k(x,x′) for x in X, x′ in X′]
```

Algorithm 18.1. The function `μ` for constructing a mean vector given a list of design points and a mean function `m`, and the function `K` for constructing a covariance matrix from a covariance function `k`.

A common kernel function is the *squared exponential kernel*, where

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right) \tag{18.9}$$

The parameter ℓ corresponds to what is called the *characteristic length-scale*, which can be thought of as the distance we have to travel in design space until the objective function value changes significantly.[5] Hence, larger values of ℓ result in smoother functions. Figure 18.2 shows functions sampled from a Gaussian process with a zero-mean function and a squared exponential kernel with different characteristic length-scales.

[5] A mathematical definition of characteristic length-scale is provided by C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

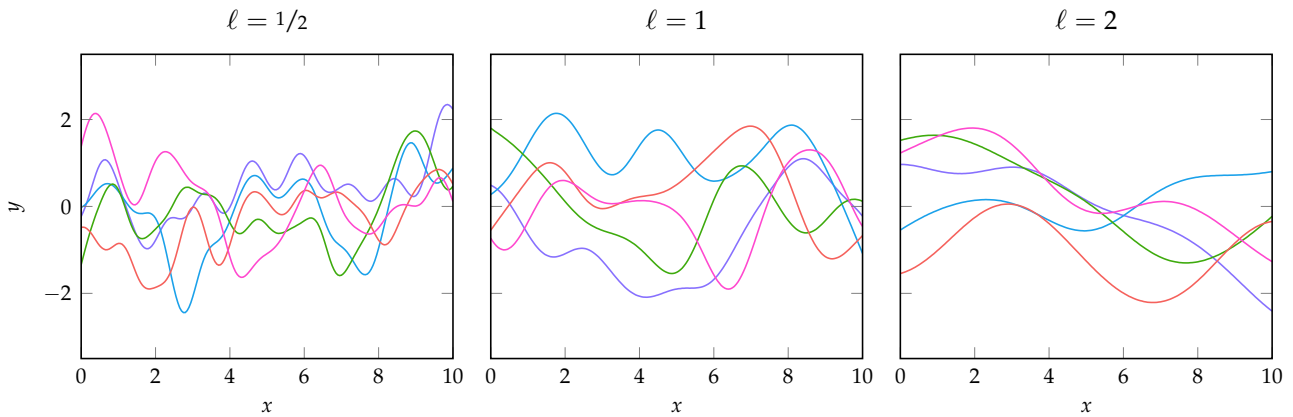


Figure 18.2. Functions sampled from Gaussian processes with squared exponential kernels.

There are many other kernel functions besides the squared exponential. Several are shown in figure 18.3. Many kernel functions use r, which is the distance between $\mathbf{x}$ and $\mathbf{x}'$. Usually the Euclidean distance is used. The *Matérn kernel* uses the *gamma function* Γ, implemented by `gamma` from the `SpecialFunctions.jl` package, and $K_\nu(x)$ is the *modified Bessel function of the second kind*, implemented by `besselk(ν,x)`. The *neural network kernel* augments each design vector with a 1 for ease of notation: $\bar{\mathbf{x}} = [1, x_1, x_2, \ldots]$ and $\bar{\mathbf{x}}' = [1, x_1', x_2', \ldots]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

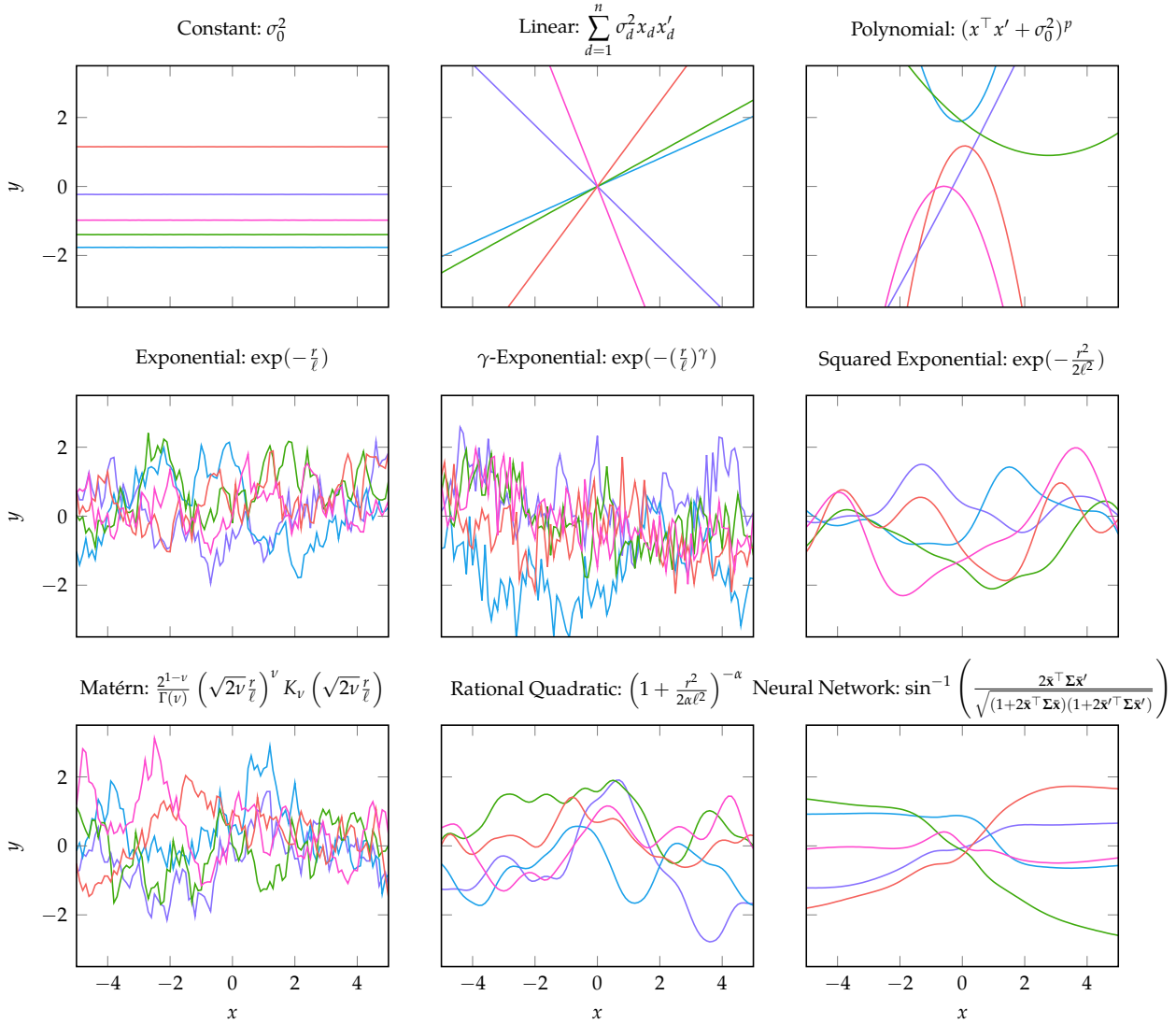


Figure 18.3. Functions sampled from Gaussian processes with different kernel functions. Shown functions are for $\sigma_0^2 = \sigma_d^2 = \ell = 1$, $p = 2$, $\gamma = \nu = \alpha = 0.5$, and $\boldsymbol{\Sigma} = \mathbf{I}$. In addition, $r = \|\mathbf{x} - \mathbf{x}'\|$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

This chapter will focus on examples of Gaussian processes with single-dimensional design spaces for ease of plotting. However, Gaussian processes can be defined over multidimensional design spaces, as illustrated in figure 18.4.

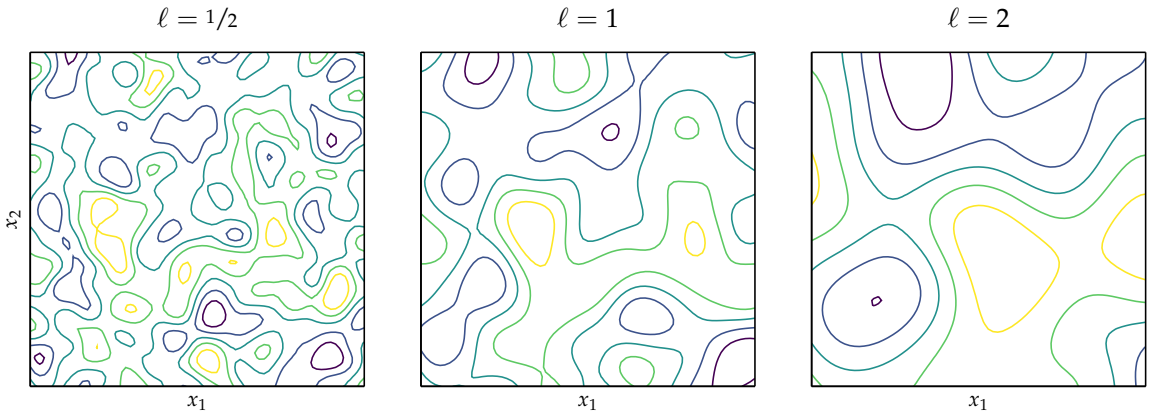


Figure 18.4. Functions sampled from a Gaussian process with zero-mean and squared-exponential kernels over a two-dimensional design space.

As we will see in section 18.5, Gaussian processes can also incorporate prior independent noise variance, denoted ν. A Gaussian process is thus defined by mean and covariance functions, prior design points and their function evaluations, and a noise variance. The associated type is given in algorithm 18.2.

```
mutable struct GaussianProcess
    m # mean
    k # covariance function
    X # design points
    y # objective values
    ν # noise variance
end
```

Algorithm 18.2. A Gaussian process is defined by a mean function `m`, a covariance function `k`, sampled design vectors `X` and their corresponding values `y`, and a noise variance `ν`.

18.3 Prediction

Gaussian processes are able to represent distributions over functions using conditional probabilities. Suppose we already have a set of points X and the corresponding $\mathbf{y}$, but we wish to predict the values $\hat{\mathbf{y}}$ at points X^*. The joint distribution

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

is

$$\begin{bmatrix}\hat{\mathbf{y}}\\ \mathbf{y}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mathbf{m}(X^*)\\ \mathbf{m}(X\ )\end{bmatrix}, \begin{bmatrix}\mathbf{K}(X^*, X^*) & \mathbf{K}(X^*, X)\\ \mathbf{K}(X\ , X^*) & \mathbf{K}(X\ , X)\end{bmatrix}\right) \tag{18.10}$$

In the equation above, we use the functions $\mathbf{m}$ and $\mathbf{K}$, which are defined as follows:

$$\mathbf{m}(X) = [m(\mathbf{x}^{(1)}), \ldots, m(\mathbf{x}^{(|X|)})] \tag{18.11}$$

$$\mathbf{K}(X, X') = \begin{bmatrix} k(\mathbf{x}^{(1)}, \mathbf{x}'^{(1)}) & \cdots & k(\mathbf{x}^{(1)}, \mathbf{x}'^{(|X'|)}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}^{(|X|)}, \mathbf{x}'^{(1)}) & \cdots & k(\mathbf{x}^{(|X|)}, \mathbf{x}'^{(|X'|)}) \end{bmatrix} \tag{18.12}$$

[6] In the language of Bayesian statistics, the posterior distribution is the distribution of possible unobserved values conditioned on observed values.

The conditional distribution is given by

$$\hat{\mathbf{y}} \mid \mathbf{y} \sim \mathcal{N}\left(\underbrace{\mathbf{m}(X^*) + \mathbf{K}(X^*, X)\mathbf{K}(X, X)^{-1}(\mathbf{y} - \mathbf{m}(X))}_{\text{mean}}, \underbrace{\mathbf{K}(X^*, X^*) - \mathbf{K}(X^*, X)\mathbf{K}(X, X)^{-1}\mathbf{K}(X, X^*)}_{\text{covariance}}\right) \tag{18.13}$$

Note that the covariance does not depend on $\mathbf{y}$. This distribution is often referred to as the *posterior distribution*.[6] A method for computing and sampling from the posterior distribution defined by a Gaussian process is given in algorithm 18.3.

```
function mvnrand(μ, Σ, inflation=1e-6)
    N = MvNormal(μ, Σ + inflation*I)
    return rand(N)
end
Base.rand(GP, X) = mvnrand(μ(X, GP.m), K(X, X, GP.k))
```

Algorithm 18.3. The function `mvnrand` samples from a multivariate Gaussian with an added inflation factor to prevent numerical issues. The method `rand` samples a Gaussian process `GP` at the given design points in matrix `X`.

The predicted mean can be written as a function of $\mathbf{x}$:

$$\hat{\mu}(\mathbf{x}) = m(\mathbf{x}) + \mathbf{K}(\mathbf{x}, X)\mathbf{K}(X, X)^{-1}(\mathbf{y} - \mathbf{m}(X)) \tag{18.14}$$

$$= m(\mathbf{x}) + \boldsymbol{\theta}^\top \mathbf{K}(X, \mathbf{x}) \tag{18.15}$$

where $\boldsymbol{\theta} = \mathbf{K}(X, X)^{-1}(\mathbf{y} - \mathbf{m}(X))$ can be computed once and reused for different values of $\mathbf{x}$. Notice the similarity to the surrogate models in the previous chapter. The value of the Gaussian process beyond the surrogate models discussed previously is that it also quantifies our uncertainty in our predictions.

The variance of the predicted mean can also be obtained as a function of $\mathbf{x}$:

$$\hat{v}(\mathbf{x}) = \mathbf{K}(\mathbf{x}, \mathbf{x}) - \mathbf{K}(\mathbf{x}, X)\mathbf{K}(X, X)^{-1}\mathbf{K}(X, \mathbf{x}) \tag{18.16}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

In some cases, it is more convenient to formulate equations in terms of the *standard deviation*, which is the square root of the variance:

$$\hat{\sigma}(\mathbf{x}) = \sqrt{\hat{\nu}(\mathbf{x})} \tag{18.17}$$

The standard deviation has the same units as the mean. From the standard deviation, we can compute the 95% *confidence region*, which is an interval containing 95% of the probability mass associated with the distribution over y given $\mathbf{x}$. For a particular $\mathbf{x}$, the 95% confidence region is given by $\hat{\mu}(\mathbf{x}) \pm 1.96\hat{\sigma}(\mathbf{x})$. One may want to use a confidence level different from 95%, but we will use 95% for the plots in this chapter. Figure 18.5 shows a plot of a confidence region associated with a Gaussian process fit to four function evaluations.

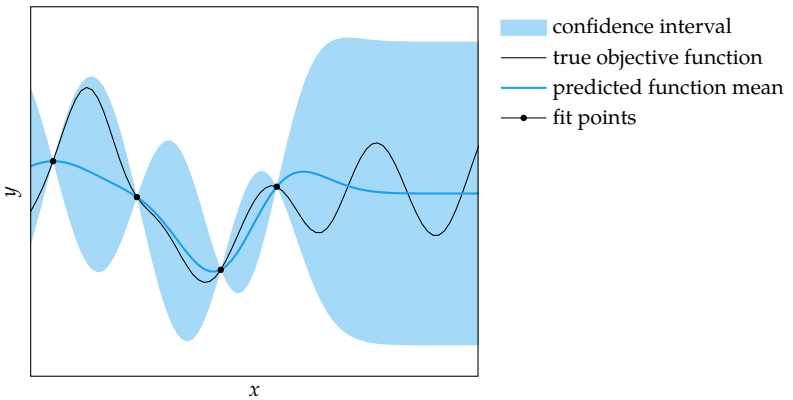


Figure 18.5. A Gaussian process using the squared exponential kernel and its 95% confidence interval. Uncertainty increases the farther we are from a data point. The expected function value approaches zero as we move far away from the data point.

18.4 Gradient Measurements

Gradient observations can be incorporated into Gaussian processes in a manner consistent with the existing Gaussian process machinery.[7] The Gaussian process is extended to include both the function value and its gradient:

$$\begin{bmatrix} \mathbf{y} \\ \nabla\mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{m}_f \\ \mathbf{m}_\nabla \end{bmatrix}, \begin{bmatrix} \mathbf{K}_{ff} & \mathbf{K}_{f\nabla} \\ \mathbf{K}_{\nabla f} & \mathbf{K}_{\nabla\nabla} \end{bmatrix} \right) \tag{18.18}$$

[7] For an overview, see for example A. O'Hagan, "Some Bayesian Numerical Analysis," in *Bayesian Statistics*, J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, eds., vol. 4, Oxford University Press, 1992, pp. 345–363.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where $\mathbf{y} \sim \mathcal{N}\left(\mathbf{m}_f, \mathbf{K}_{ff}\right)$ is a traditional Gaussian process, $\mathbf{m}_\nabla$ is a mean function for the gradient,[8] $\mathbf{K}_{f\nabla}$ is the covariance matrix between function values and gradients, $\mathbf{K}_{\nabla f}$ is the covariance matrix between function gradients and values, and $\mathbf{K}_{\nabla\nabla}$ is the covariance matrix between function gradients.

[8] Like the mean of the function value, $\mathbf{m}_\nabla$ is often zero.

These covariance matrices are constructed using covariance functions. The linearity of Gaussians causes these covariance functions to be related:

$$k_{ff}(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}, \mathbf{x}') \tag{18.19}$$
$$k_{\nabla f}(\mathbf{x}, \mathbf{x}') = \nabla_{\mathbf{x}} k(\mathbf{x}, \mathbf{x}') \tag{18.20}$$
$$k_{f\nabla}(\mathbf{x}, \mathbf{x}') = \nabla_{\mathbf{x}'} k(\mathbf{x}, \mathbf{x}') \tag{18.21}$$
$$k_{\nabla\nabla}(\mathbf{x}, \mathbf{x}') = \nabla_{\mathbf{x}} \nabla_{\mathbf{x}'} k(\mathbf{x}, \mathbf{x}') \tag{18.22}$$

Example 18.2 uses these relations to derive the higher-order covariance functions for a particular kernel.

Example 18.2. Deriving covariance functions for a Gaussian process with gradient observations.

Consider the squared exponential covariance function

$$k_{ff}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2\right)$$

We can use equations (18.19) to (18.22) to obtain the other covariance functions necessary for using Gaussian processes with gradient information:

$$k_{\nabla f}(\mathbf{x}, \mathbf{x}')_i = -(\mathbf{x}_i - \mathbf{x}'_i) \exp\left(-\frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2\right)$$
$$k_{\nabla\nabla}(\mathbf{x}, \mathbf{x}')_{ij} = \left((i = j) - (\mathbf{x}_i - \mathbf{x}'_i)(\mathbf{x}_j - \mathbf{x}'_j)\right) \exp\left(-\frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|^2\right)$$

As a reminder, Boolean expressions, such as $(i = j)$, return 1 if true and 0 if false.

Prediction can be accomplished in the same manner as with a traditional Gaussian process. We first construct the joint distribution

$$\begin{bmatrix} \hat{\mathbf{y}} \\ \mathbf{y} \\ \nabla\mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{m}_f(X^*) \\ \mathbf{m}_f(X\ ) \\ \mathbf{m}_\nabla(X\ ) \end{bmatrix}, \begin{bmatrix} \mathbf{K}_{ff}(X^*, X^*) & \mathbf{K}_{ff}(X^*, X\ ) & \mathbf{K}_{f\nabla}(X^*, X\ ) \\ \mathbf{K}_{ff}(X\ , X^*) & \mathbf{K}_{ff}(X\ , X\ ) & \mathbf{K}_{f\nabla}(X\ , X\ ) \\ \mathbf{K}_{\nabla f}(X\ , X^*) & \mathbf{K}_{\nabla f}(X\ , X\ ) & \mathbf{K}_{\nabla\nabla}(X\ , X\ ) \end{bmatrix}\right) \tag{18.23}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

For a Gaussian process over n-dimensional design vectors given m pairs of function and gradient evaluations and ℓ query points, the covariance blocks have the following dimensions:

$$\begin{matrix} \ell \times \ell & \ell \times m & \ell \times nm \\ m \times \ell & m \times m & m \times nm \\ nm \times \ell & nm \times m & nm \times nm \end{matrix} \tag{18.24}$$

Example 18.3 constructs such a covariance matrix.

Example 18.3. Constructing the covariance matrix for a Gaussian process with gradient observations.

Suppose we have evaluated a function and its gradient at two locations, $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$, and we wish to predict the function value at $\hat{\mathbf{x}}$. We can infer the joint distribution over $\hat{\mathbf{y}}$, $\mathbf{y}$, and $\nabla\mathbf{y}$ using a Gaussian process. The covariance matrix is:

$$\begin{bmatrix} k_{ff}(\hat{\mathbf{x}},\hat{\mathbf{x}}) & k_{ff}(\hat{\mathbf{x}},\mathbf{x}^{(1)}) & k_{ff}(\hat{\mathbf{x}},\mathbf{x}^{(2)}) & k_{f\nabla}(\hat{\mathbf{x}},\mathbf{x}^{(1)})_1 & k_{f\nabla}(\hat{\mathbf{x}},\mathbf{x}^{(1)})_2 & k_{f\nabla}(\hat{\mathbf{x}},\mathbf{x}^{(2)})_1 & k_{f\nabla}(\hat{\mathbf{x}},\mathbf{x}^{(2)})_2 \\ k_{ff}(\mathbf{x}^{(1)},\hat{\mathbf{x}}) & k_{ff}(\mathbf{x}^{(1)},\mathbf{x}^{(1)}) & k_{ff}(\mathbf{x}^{(1)},\mathbf{x}^{(2)}) & k_{f\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_1 & k_{f\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_2 & k_{f\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_1 & k_{f\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_2 \\ k_{ff}(\mathbf{x}^{(2)},\hat{\mathbf{x}}) & k_{ff}(\mathbf{x}^{(2)},\mathbf{x}^{(1)}) & k_{ff}(\mathbf{x}^{(2)},\mathbf{x}^{(2)}) & k_{f\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_1 & k_{f\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_2 & k_{f\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_1 & k_{f\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_2 \\ k_{\nabla f}(\mathbf{x}^{(1)},\hat{\mathbf{x}})_1 & k_{\nabla f}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_1 & k_{\nabla f}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_1 & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_{11} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_{12} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_{11} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_{12} \\ k_{\nabla f}(\mathbf{x}^{(1)},\hat{\mathbf{x}})_2 & k_{\nabla f}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_2 & k_{\nabla f}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_2 & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_{21} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(1)})_{22} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_{21} & k_{\nabla\nabla}(\mathbf{x}^{(1)},\mathbf{x}^{(2)})_{22} \\ k_{\nabla f}(\mathbf{x}^{(2)},\hat{\mathbf{x}})_1 & k_{\nabla f}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_1 & k_{\nabla f}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_1 & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_{11} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_{12} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_{11} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_{12} \\ k_{\nabla f}(\mathbf{x}^{(2)},\hat{\mathbf{x}})_2 & k_{\nabla f}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_2 & k_{\nabla f}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_2 & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_{21} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(1)})_{22} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_{21} & k_{\nabla\nabla}(\mathbf{x}^{(2)},\mathbf{x}^{(2)})_{22} \end{bmatrix}$$

The conditional distribution follows the same Gaussian relations as in equation (18.13):

$$\hat{\mathbf{y}} \mid \mathbf{y}, \nabla\mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_\nabla, \boldsymbol{\Sigma}_\nabla) \tag{18.25}$$

where:

$$\boldsymbol{\mu}_\nabla = \mathbf{m}_f(X^*) + \begin{bmatrix} \mathbf{K}_{ff}(X\ , X^*) \\ \mathbf{K}_{\nabla f}(X\ , X^*) \end{bmatrix}^\top \begin{bmatrix} \mathbf{K}_{ff}(X\ , X\ ) & \mathbf{K}_{f\nabla}(X\ , X\ ) \\ \mathbf{K}_{\nabla f}(X\ , X\ ) & \mathbf{K}_{\nabla\nabla}(X\ , X\ ) \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{y} - \mathbf{m}_f(X) \\ \nabla\mathbf{y} - \mathbf{m}_\nabla(X) \end{bmatrix} \tag{18.26}$$

$$\boldsymbol{\Sigma}_\nabla = \mathbf{K}_{ff}(X^*, X^*) - \begin{bmatrix} \mathbf{K}_{ff}(X\ , X^*) \\ \mathbf{K}_{\nabla f}(X\ , X^*) \end{bmatrix}^\top \begin{bmatrix} \mathbf{K}_{ff}(X\ , X\ ) & \mathbf{K}_{f\nabla}(X\ , X\ ) \\ \mathbf{K}_{\nabla f}(X\ , X\ ) & \mathbf{K}_{\nabla\nabla}(X\ , X\ ) \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{K}_{ff}(X\ , X^*) \\ \mathbf{K}_{\nabla f}(X\ , X^*) \end{bmatrix} \tag{18.27}$$

The regions obtained when including gradient observations are compared to those without gradient observations in figure 18.6.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

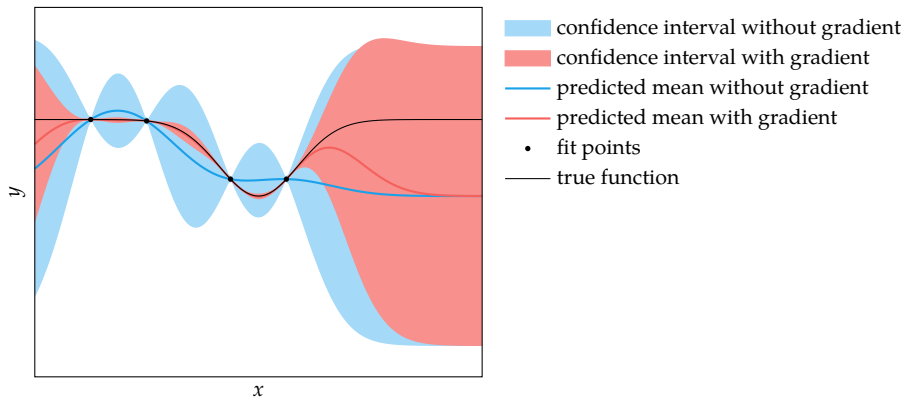


Figure 18.6. Gaussian processes with and without gradient information using squared exponential kernels. Incorporating gradient information can significantly reduce the confidence intervals.

18.5 Noisy Measurements

So far we have assumed that the objective function f is deterministic. In practice, however, evaluations of f may include measurement noise, experimental error, or numerical roundoff.

We can model noisy evaluations as $y = f(\mathbf{x}) + z$, where f is deterministic but z is zero-mean Gaussian noise, $z \sim \mathcal{N}(0, \nu)$. The variance of the noise ν can be adjusted to control the uncertainty.[9]

[9] The techniques covered in section 17.5 can be used to tune the variance of the noise.

The new joint distribution is:

$$\begin{bmatrix}\hat{\mathbf{y}}\\ \mathbf{y}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mathbf{m}(X^*)\\ \mathbf{m}(X\ )\end{bmatrix}, \begin{bmatrix}\mathbf{K}(X^*, X^*) & \mathbf{K}(X^*, X)\\ \mathbf{K}(X\ , X^*) & \mathbf{K}(X\ , X) + \nu\mathbf{I}\end{bmatrix}\right) \tag{18.28}$$

with conditional distribution:

$$\hat{\mathbf{y}} \mid \mathbf{y}, \nu \sim \mathcal{N}(\boldsymbol{\mu}^*, \boldsymbol{\Sigma}^*) \tag{18.29}$$

$$\boldsymbol{\mu}^* = \mathbf{m}(X^*) + \mathbf{K}(X^*, X)(\mathbf{K}(X, X) + \nu\mathbf{I})^{-1}(\mathbf{y} - \mathbf{m}(X)) \tag{18.30}$$

$$\boldsymbol{\Sigma}^* = \mathbf{K}(X^*, X^*) - \mathbf{K}(X^*, X)(\mathbf{K}(X, X) + \nu\mathbf{I})^{-1}\mathbf{K}(X, X^*) \tag{18.31}$$

As the equations above show, accounting for Gaussian noise is straightforward and the posterior distribution can be computed analytically. Figure 18.7 shows a noisy Gaussian process. Algorithm 18.4 implements prediction for Gaussian processes with noisy measurements.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

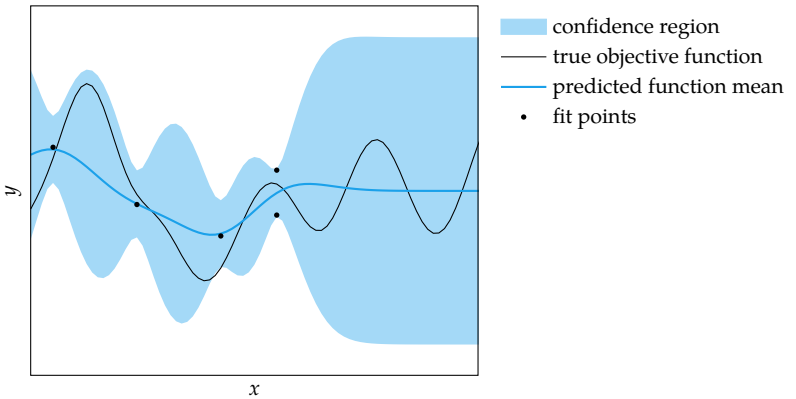


Figure 18.7. A noisy Gaussian process using a squared exponential kernel.

```
function predict(GP, X_pred)
    m, k, ν = GP.m, GP.k, GP.ν
    tmp = K(X_pred, GP.X, k) / (K(GP.X, GP.X, k) + ν*I)
    μₚ = μ(X_pred, m) + tmp*(GP.y - μ(GP.X, m))
    S = K(X_pred, X_pred, k) - tmp*K(GP.X, X_pred, k)
    νₚ = diag(S) .+ eps() # eps prevents numerical issues
    return (μₚ, νₚ)
end
```

Algorithm 18.4. A method for obtaining the predicted means and variances in f under a Gaussian process. The method takes a Gaussian process `GP` and a list of points `X_pred` at which to evaluate the prediction. It returns the mean and variance at each evaluation point.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

18.6 Fitting Gaussian Processes

The choice of kernel and parameters has a large effect on the form of the Gaussian process between evaluated design points. Kernels and their parameters can be chosen using cross-validation introduced in the previous chapter. Instead of minimizing the squared error on the test data, we maximize the likelihood of the data.[10] That is, we seek the parameters θ that maximize the probability of the function values, $p(\mathbf{y} \mid X, \theta)$. The likelihood of the data is the probability that the observed points were drawn from the model. Equivalently, we can maximize the *log likelihood*, which is generally preferable because multiplying small probabilities in the likelihood calculation can produce extremely small values. Given a dataset $\mathcal{D}$ with n entries, the log likelihood is given by

[10] Alternatively, we could maximize the pseudolikelihood as discussed by C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006.

$$\log p(\mathbf{y} \mid X, \nu, \theta) = -\frac{n}{2}\log 2\pi - \frac{1}{2}\log|\mathbf{K}_\theta(X, X) + \nu\mathbf{I}| - \frac{1}{2}(\mathbf{y} - \mathbf{m}_\theta(X))^\top(\mathbf{K}_\theta(X, X) + \nu\mathbf{I})^{-1}(\mathbf{y} - \mathbf{m}_\theta(X)) \tag{18.32}$$

where the mean and covariance functions are parameterized by θ.

Let us assume a zero mean such that $\mathbf{m}_\theta(X) = \mathbf{0}$ and θ refers only to the parameters for the Gaussian process covariance function. We can arrive at a *maximum likelihood estimate* by gradient ascent. The gradient is then given by

$$\frac{\partial}{\partial\theta_j}\log p(\mathbf{y} \mid X, \theta) = \frac{1}{2}\mathbf{y}^\top\boldsymbol{\Sigma}_\theta^{-1}\frac{\partial\mathbf{K}_\theta}{\partial\theta_j}\boldsymbol{\Sigma}_\theta^{-1}\mathbf{y} - \frac{1}{2}\text{tr}\left(\boldsymbol{\Sigma}_\theta^{-1}\frac{\partial\mathbf{K}_\theta}{\partial\theta_j}\right) \tag{18.33}$$

where $\boldsymbol{\Sigma}_\theta = \mathbf{K}_\theta(X, X) + \nu\mathbf{I}$. Above, we use the matrix derivative relations

$$\frac{\partial\mathbf{K}_\theta^{-1}}{\partial\theta_j} = -\mathbf{K}_\theta^{-1}\frac{\partial\mathbf{K}_\theta}{\partial\theta_j}\mathbf{K}_\theta^{-1} \tag{18.34}$$

$$\frac{\partial\log|\mathbf{K}_\theta|}{\partial\theta_j} = \text{tr}\left(\mathbf{K}_\theta^{-1}\frac{\partial\mathbf{K}_\theta}{\partial\theta_j}\right) \tag{18.35}$$

where $\text{tr}(\mathbf{A})$ denotes the *trace* of a matrix $\mathbf{A}$, defined to be the sum of the elements on the main diagonal.

18.7 Summary

- Gaussian processes are probability distributions over functions.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- The choice of kernel affects the smoothness of the functions sampled from a Gaussian process.
- The multivariate normal distribution has analytic conditional and marginal distributions.
- We can compute the mean and standard deviation of our prediction of an objective function at a particular design point given a set of past evaluations.
- We can incorporate gradient observations to improve our predictions of the objective value and its gradient.
- We can incorporate measurement noise into a Gaussian process.
- We can fit the parameters of a Gaussian process using maximum likelihood.

18.8 Exercises

Exercise 18.1. Gaussian processes will grow in complexity during the optimization process as more samples accumulate. How can this be an advantage over models based on regression?

Solution: Gaussian processes are *nonparametric*, whereas linear regression models are *parametric*. This means that the number of degrees of freedom of the model grows with the amount of data, allowing the Gaussian process to maintain a balance between bias and variance during the optimization process.

Exercise 18.2. How does the computational complexity of prediction with a Gaussian process increase with the number of data points m?

Solution: Obtaining the conditional distribution of a Gaussian process requires solving equation (18.13). The most expensive operation is inverting the $m \times m$ matrix $\mathbf{K}(X, X)$, which is $O(m^3)$.

Exercise 18.3. Consider the function $f(x) = \sin(x)/\left(x^2 + 1\right)$ over $[-5, 5]$. Plot the 95% confidence bounds for a Gaussian process with derivative information fitted to the evaluations at $\{-5, -2.5, 0, 2.5, 5\}$. What is the maximum standard deviation in the predicted distribution within $[-5, 5]$? How many function evaluations, evenly spaced over the domain, are needed such that a Gaussian process without derivative information achieves the same maximum predictive standard deviation?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

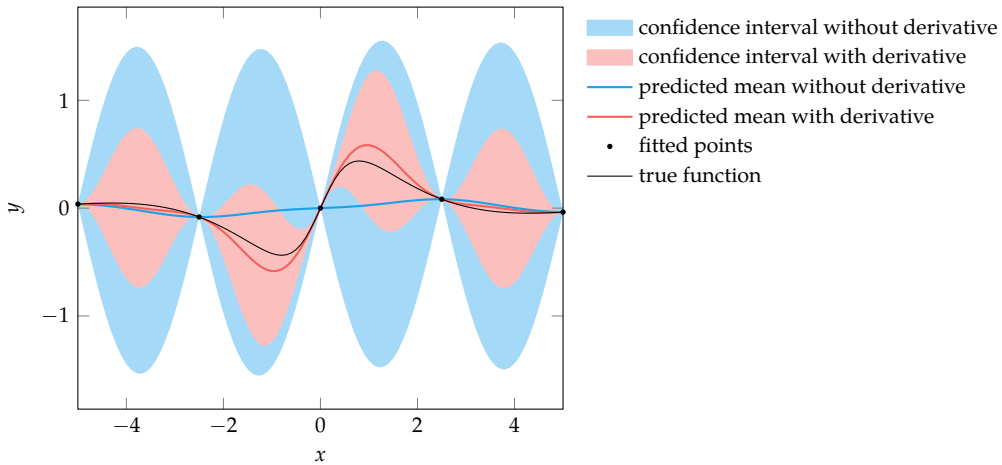
Assume zero-mean functions and noise-free observations, and use the covariance functions:

$$k_{ff}(x, x') = \exp\left(-\frac{1}{2}\|x - x'\|_2^2\right)$$
$$k_{\nabla f}(x, x') = (x' - x)\exp\left(-\frac{1}{2}\|x - x'\|_2^2\right)$$
$$k_{\nabla\nabla}(x, x') = (1 - (x - x')^2)\exp(-\frac{1}{2}\|x - x'\|_2^2)$$

Solution: The derivative of f is

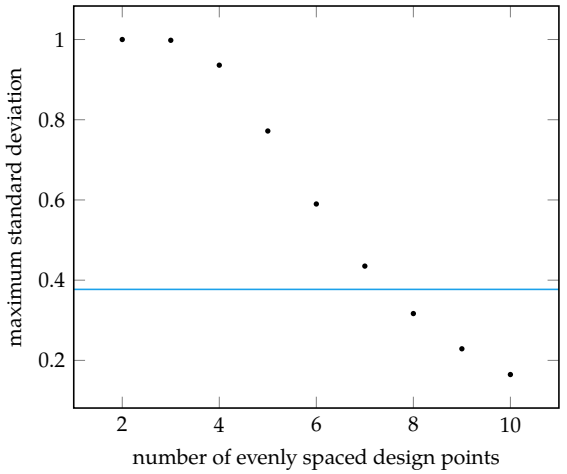
$$\frac{(x^2 + 1)\cos(x) - 2x\sin(x)}{(x^2 + 1)^2}$$

Below we plot the predictive distribution for Gaussian processes with and without derivative information. The maximum standard deviation in the predicted distribution over $[-5, 5]$ for the Gaussian process with derivative information is approximately 0.377 at $x \approx \pm 3.8$.



Incorporating derivative information significantly decreases the confidence interval because more information is available to inform the prediction. Below we plot the maximum standard deviation in the predicted distribution over $[-5, 5]$ for Gaussian processes without derivative information with a varying number of evenly spaced evaluations. At least eight points are needed in order to outperform the Gaussian process with derivative information.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Exercise 18.4. Derive the relation $k_{f\nabla}(\mathbf{x}, \mathbf{x}')_i = \text{cov}\left(f(\mathbf{x}), \frac{\partial}{\partial x_i'} f(\mathbf{x}')\right) = \frac{\partial}{\partial x_i'} k_{ff}(\mathbf{x}, \mathbf{x}')$.

Solution: This can be derived according to:

$$\begin{aligned}
k_{f\nabla}(\mathbf{x}, \mathbf{x}')_i &= \text{cov}\left(f(\mathbf{x}), \frac{\partial}{\partial x_i'} f(\mathbf{x}')\right) \\
&= \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[f(\mathbf{x})])\left(\frac{\partial}{\partial x_i'} f(\mathbf{x}') - \mathbb{E}\left[\frac{\partial}{\partial x_i'} f(\mathbf{x}')\right]\right)\right] \\
&= \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[f(\mathbf{x})])\left(\frac{\partial}{\partial x_i'} f(\mathbf{x}') - \frac{\partial}{\partial x_i'} \mathbb{E}\left[f(\mathbf{x}')\right]\right)\right] \\
&= \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[f(\mathbf{x})]) \frac{\partial}{\partial x_i'} \left(f(\mathbf{x}') - \mathbb{E}\left[f(\mathbf{x}')\right]\right)\right] \\
&= \frac{\partial}{\partial x_i'} \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[f(\mathbf{x})])\left(f(\mathbf{x}') - \mathbb{E}\left[f(\mathbf{x}')\right]\right)\right] \\
&= \frac{\partial}{\partial x_i'} \text{cov}\left(f(\mathbf{x}), f(\mathbf{x}')\right) \\
&= \frac{\partial}{\partial x_i'} k_{ff}(\mathbf{x}, \mathbf{x}')
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where we have used $\mathbb{E}\left[\frac{\partial}{\partial x} f\right] = \frac{\partial}{\partial x}\mathbb{E}[f]$. We can convince ourselves that this is true:

$$\begin{aligned}
\mathbb{E}\left[\frac{\partial}{\partial x} f\right] &= \mathbb{E}\left[\lim_{h\to 0} \frac{f(x+h) - f(x)}{h}\right] \\
&= \lim_{h\to 0} \mathbb{E}\left[\frac{f(x+h) - f(x)}{h}\right] \\
&= \lim_{h\to 0} \frac{1}{h}(\mathbb{E}[f(x+h)] - \mathbb{E}[f(x)]) \\
&= \frac{\partial}{\partial x}\mathbb{E}[f(x)]
\end{aligned}$$

provided that the objective function is differentiable.

Exercise 18.5. Suppose we have a multivariate Gaussian distribution over two variables a and b. Show that the variance of the conditional distribution over a given b is no greater than the variance of the marginal distribution over a. Does this make intuitive sense?

Solution: Let us write the joint Gaussian distribution as:

$$\begin{bmatrix} a \\ b \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_a \\ \mu_b \end{bmatrix}, \begin{bmatrix} \nu_a & \nu_c \\ \nu_c & \nu_b \end{bmatrix}\right)$$

The marginal distribution over a is $\mathcal{N}(\mu_a, \nu_a)$, which has variance ν_a. The conditional distribution for a has variance $\nu_a - \nu_c^2/\nu_b$. We know ν_b must be positive in order for the original covariance matrix to be positive definite. Thus, ν_c^2/ν_b is positive and $\nu_a - \nu_c^2/\nu_b \leq \nu_a$.

It is intuitive that the conditional distribution has no greater variance than the marginal distribution because the conditional distribution incorporates more information about a. If a and b are correlated, then knowing the value of b informs us about the value of a and decreases our uncertainty.

Exercise 18.6. Suppose we observe many outliers while sampling; that is, we observe samples that do not fall within the confidence interval given by the Gaussian process. This means the probabilistic model we chose is not appropriate. What can we do?

Solution: We can tune the parameters to our kernel function or switch kernel functions using generalization error estimation or by maximizing the likelihood of the observed data.

Exercise 18.7. Equation (18.10) provides the joint distribution over observed evaluations $\mathbf{y}$ at points X and the evaluations $\hat{\mathbf{y}}$ at new points X^*. Suppose we wish to produce a multifidelity surrogate model (section 17.6) over both low-fidelity evaluations $\mathbf{y}_\ell$ at points X_ℓ and high-fidelity evaluations $\mathbf{y}_h$ at points X_h using a single Gaussian process. We assume the low- and high-fidelity points are in a common design space. Modify equation (18.10) to provide a joint distribution over $\hat{\mathbf{y}}_h$, $\mathbf{y}_\ell$, and $\mathbf{y}_h$, and then write the predicted mean as a function of $\mathbf{x}$. Use the following three kernel functions to capture the expected covariance between designs:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- $k_{\ell\ell}(\mathbf{x}, \mathbf{x}')$ for the covariance between two low-fidelity measurements,
- $k_{hh}(\mathbf{x}, \mathbf{x}')$ for the covariance between two high-fidelity measurements, and
- $k_{\ell h}(\mathbf{x}, \mathbf{x}')$ for the covariance between a low-fidelity and a high-fidelity measurement.

Solution: We begin by writing a joint distribution over our low-fidelity and high-fidelity data:

$$\begin{bmatrix}\mathbf{y}_\ell \\ \mathbf{y}_h\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mathbf{m}_\ell(X_\ell) \\ \mathbf{m}_h(X_h)\end{bmatrix}, \begin{bmatrix}\mathbf{K}_{\ell\ell}(X_\ell, X_\ell) & \mathbf{K}_{\ell h}(X_\ell, X_h) \\ \mathbf{K}_{h\ell}(X_h, X_\ell) & \mathbf{K}_{hh}(X_h, X_h)\end{bmatrix}\right)$$

The mean functions $m_\ell(\mathbf{x})$ and $m_h(\mathbf{x})$ represent our mean predictions for the low- and high-fidelity values in the absence of nearby supporting data, and can typically be obtained using priors or by fitting (non-probabilistic) surrogate models.

We can then expand this joint distribution to include high-fidelity values we wish to predict $\hat{\mathbf{y}}_h$ at points X^*:

$$\begin{bmatrix}\hat{\mathbf{y}}_h \\ \mathbf{y}_\ell \\ \mathbf{y}_h\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mathbf{m}_h(X^*) \\ \mathbf{m}_\ell(X_\ell) \\ \mathbf{m}_h(X_h)\end{bmatrix}, \begin{bmatrix}\mathbf{K}_{hh}(X^*, X^*) & \mathbf{K}_{\ell h}(X^*, X_\ell) & \mathbf{K}_{hh}(X^*, X_h) \\ \mathbf{K}_{\ell h}(X_\ell, X^*) & \mathbf{K}_{\ell\ell}(X_\ell, X_\ell) & \mathbf{K}_{\ell h}(X_\ell, X_h) \\ \mathbf{K}_{hh}(X_h, X^*) & \mathbf{K}_{\ell h}(X_h, X_\ell) & \mathbf{K}_{hh}(X_h, X_h)\end{bmatrix}\right)$$

This joint distribution is analogous to equation (18.10), and very closely reflects equation (18.23). Notice that because we are inferring high-fidelity data, we treat the inferred measurements as high-fidelity with respect to the mean and kernel functions.

The conditional distribution has a mean:

$$\mathbf{m}_h(X^*) + \begin{bmatrix}\mathbf{K}_{\ell h}(X^*, X_\ell) & \mathbf{K}_{hh}(X^*, X_h)\end{bmatrix} \begin{bmatrix}\mathbf{K}_{\ell\ell}(X_\ell, X_\ell) & \mathbf{K}_{\ell h}(X_\ell, X_h) \\ \mathbf{K}_{h\ell}(X_h, X_\ell) & \mathbf{K}_{hh}(X_h, X_h)\end{bmatrix}^{-1} \left(\begin{bmatrix}\mathbf{y}_\ell \\ \mathbf{y}_h\end{bmatrix} - \begin{bmatrix}\mathbf{m}_\ell(X_\ell) \\ \mathbf{m}_h(X_h)\end{bmatrix}\right)$$

The predicted high-fidelity mean as a function of $\mathbf{x}$ is then given by:

$$\hat{\mu}_h(\mathbf{x}) = \mathbf{m}_h(\mathbf{x}) + \begin{bmatrix}\mathbf{K}_{\ell h}(\mathbf{x}, X_\ell) & \mathbf{K}_{hh}(\mathbf{x}, X_h)\end{bmatrix} \begin{bmatrix}\mathbf{K}_{\ell\ell}(X_\ell, X_\ell) & \mathbf{K}_{\ell h}(X_\ell, X_h) \\ \mathbf{K}_{h\ell}(X_h, X_\ell) & \mathbf{K}_{hh}(X_h, X_h)\end{bmatrix}^{-1} \begin{bmatrix}\mathbf{y}_\ell - \mathbf{m}_\ell(X_\ell) \\ \mathbf{y}_h - \mathbf{m}_h(X_h)\end{bmatrix}$$

Exercise 18.8. Fit a Gaussian process to the following data

```
xs = [-0.8,  -0.6,  -0.4,  -0.2,   0.0,   0.2,   0.4,   0.6,   0.8,   1.0]
ys = [-0.463, -0.148, 0.277, 0.512, 0.425, 0.052, -0.404, -0.644, -0.996, -0.360]
```

using a mean function based on a 5th-order polynomial fit and a squared exponential kernel with $\ell = 0.25$. Plot the mean prediction of your model and its 95% confidence interval.

Solution: Linear regression with 5th-order polynomial bases on the given data produces the weights:

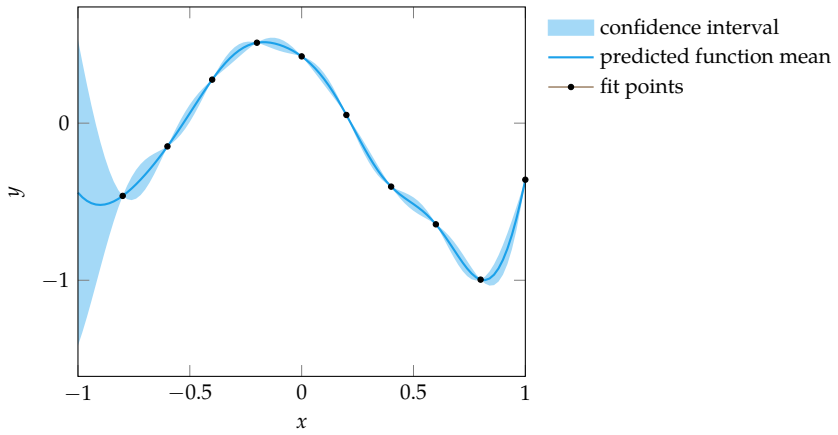
```
θ = [0.419, -0.991, -3.308, 1.324, 2.471, -0.298]
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We then construct our Gaussian process using the mean function

$$m(x) = \theta_1 + \theta_2 x + \theta_3 x^2 + \theta_4 x^3 + \theta_5 x^4 + \theta_6 x^5$$

and a squared exponential kernel with $\ell = 0.25$:



Exercise 18.9. Suppose the Gaussian process in exercise 18.8 was fit on data from a low-fidelity dataset, but we also had additional data from a high-fidelity dataset:

```
xs = [-0.7,    0.5,    0.9]
ys = [-0.220, -0.737, -1.449]
```

Extend that Gaussian process into a multifidelity Gaussian process by incorporating separate estimates for the high-fidelity data. Again use linear regression to determine a mean function $m_h(x)$, but use $m_\ell(x)$ as one of the bases in addition to a 2nd-order polynomial:[11]

$$m_h(x) = \theta_1 m_\ell(x) + \theta_2 + \theta_3 x + \theta_4 x^2$$

Under this scheme, the high-fidelity model is explicitly correlated with the low-fidelity model through the learned weight θ_1. We have $k_{\ell h}(x, x') = \theta_1 k_{\ell\ell}(x, x')$. Similarly, the covariance of the high-fidelity function is at least $k_{hh}(x, x') = \theta_1^2 k_{\ell\ell}(x, x')$.

Use your mean functions and exercise 18.7 to construct a multifidelity Gaussian process using squared exponential kernels with $\ell = 0.25$. Choose a suitable scaling for the high-fidelity kernel function.

[11] It is common practice to represent the high-fidelity estimates in multifidelity surrogate optimization as adjustments to a low-fidelity model. In this context, the additional bases are called a *corrector function*.

Solution: Linear regression to learn the high-fidelity basis weights yields:

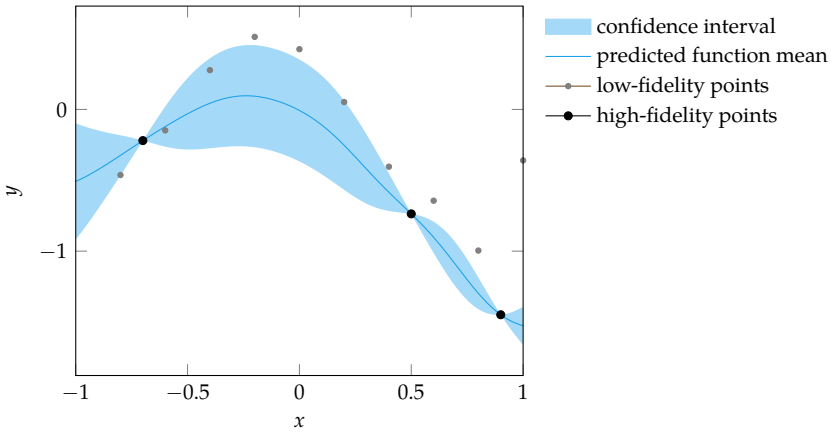
```
θhi = [0.256, -0.112, -0.536, -0.820]
```

Below we show a multifidelity Gaussian process using these mean functions. The low-fidelity kernel $k_{\ell\ell}$ is a squared exponential kernel with $\ell = 0.25$, and the cross-term kernels are thus:

$$k_{\ell h}(\mathbf{x}, \mathbf{x}') = k_{h\ell}(\mathbf{x}, \mathbf{x}') = 0.256 k_{\ell\ell}(\mathbf{x}, \mathbf{x}')$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We used a high-fidelity kernel of $k_{hh}(\mathbf{x}, \mathbf{x}') = 0.1k_{\ell\ell}(\mathbf{x}, \mathbf{x}')$. Varying the scaling term will change the confidence intervals.



We can see how the high-fidelity model uses information from the low-fidelity model to bend upward in the middle. It does not directly lie on the low-fidelity points, and even learns to lie offset from the low-fidelity model on the right side of the plot.

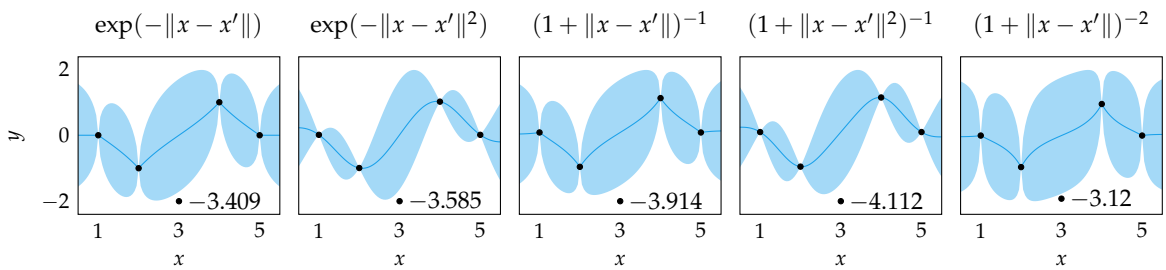
Exercise 18.10. Consider model selection for the function evaluation pairs (x, y):

$$\{(1,0), (2,-1), (3,-2), (4,1), (5,0)\}$$

Use leave-one-out cross-validation to select the kernel that maximizes the likelihood of predicting the withheld pair given a Gaussian process over the other pairs in the fold. Assume zero mean with no noise. Select from the kernels:

$$\exp(-\|x - x'\|) \qquad \exp(-\|x - x'\|^2) \qquad (1 + \|x - x'\|)^{-1} \qquad (1 + \|x - x'\|^2)^{-1} \qquad (1 + \|x - x'\|)^{-2}$$

Solution: Maximizing the product of the likelihoods is equivalent to maximizing the sum of the log likelihoods. Here are the log likelihoods of the third point given the other points using each kernel:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Computing these values over all five folds yields the total log likelihoods:

$$\begin{aligned}
\exp(-\|x - x'\|) &\to -8.688 \\
\exp(-\|x - x'\|^2) &\to -9.010 \\
(1 + \|x - x'\|)^{-1} &\to -9.579 \\
(1 + \|x - x'\|^2)^{-1} &\to -10.195 \\
(1 + \|x - x'\|)^{-2} &\to -8.088
\end{aligned}$$

It follows that the kernel that maximizes the leave-one-out cross-validated likelihood is the kernel $(1 + \|x - x'\|)^{-2}$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

19 Surrogate Optimization

The previous chapter explained how to use a probabilistic surrogate model, in particular a Gaussian process, to infer Bayesian posterior probability distributions over the true objective function. These distributions can be used to guide an optimization process toward better design points in a process known as *Bayesian optimization*.[1] This chapter outlines several common techniques for choosing which design point to evaluate next. The techniques we discuss here greedily optimize various metrics.[2] We will also discuss how surrogate models can be used to optimize an objective measure in a safe manner.

[1] R. Garnett, *Bayesian Optimization*. Cambridge University Press, 2023.

[2] An alternative to greedy optimization is to frame the problem as a *partially observable Markov decision process* and plan ahead some number of steps as outlined by M. Toussaint, "The Bayesian Search Game," in *Theory and Principled Methods for the Design of Metaheuristics*, Y. Borenstein and A. Moraglio, eds. Springer, 2014, pp. 129–144. See also R. Lam, K. Willcox, and D. H. Wolpert, "Bayesian Optimization with a Finite Budget: An Approximate Dynamic Programming Approach," in *Advances in Neural Information Processing Systems (NIPS)*, 2016.

19.1 Prediction-Based Exploration

In *prediction-based exploration*, we select the minimizer of the surrogate function. An example of this approach is the quadratic fit search that we discussed earlier in section 3.5. With quadratic fit search, we use a quadratic surrogate model to fit the last three bracketing points and then select the point at the minimum of the quadratic function.

If we use a Gaussian process surrogate model, prediction-based optimization has us select the minimizer of the mean function

$$\mathbf{x}^{(m+1)} = \arg\min_{\mathbf{x} \in \mathcal{X}} \hat{\mu}(\mathbf{x}) \tag{19.1}$$

where $\hat{\mu}(\mathbf{x})$ is the predicted mean of a Gaussian process at a design point $\mathbf{x}$ based on the previous m design points. The process is illustrated in figure 19.1.

Prediction-based optimization does not take uncertainty into account, and new samples can be generated very close to existing samples. Sampling at locations where we are already confident in the objective value is a waste of function evaluations.

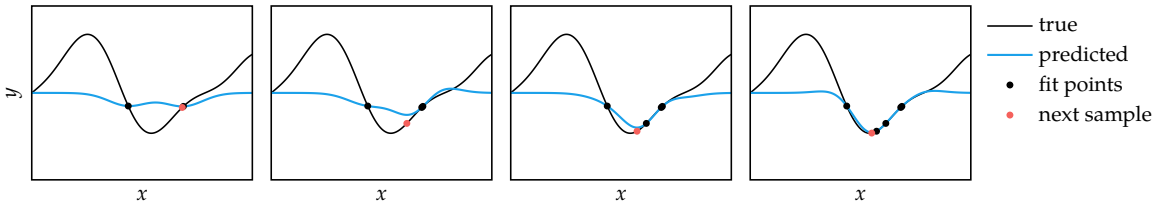


Figure 19.1. Prediction-based optimization selects the point that minimizes the mean of the objective function.

19.2 Error-Based Exploration

Error-based exploration seeks to increase confidence in the true function. A Gaussian process can tell us both the mean and standard deviation at every point. A large standard deviation indicates low confidence, so error-based exploration samples at design points with maximum uncertainty.

The next sample point is:

$$\mathbf{x}^{(m+1)} = \arg\max_{\mathbf{x} \in \mathcal{X}} \hat{\sigma}(\mathbf{x}) \tag{19.2}$$

where $\hat{\sigma}(\mathbf{x})$ is the standard deviation of a Gaussian process at a design point $\mathbf{x}$ based on the previous m design points. The process is illustrated in figure 19.2.

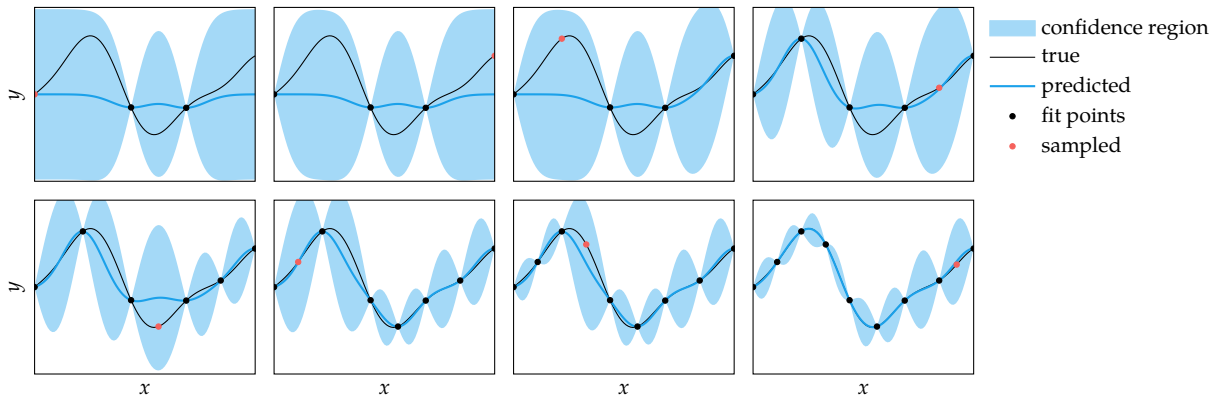


Figure 19.2. Error-based exploration selects a point with maximal uncertainty.

Gaussian processes are often defined over all of $\mathbb{R}^n$. Optimization problems with unbounded feasible sets will always have high uncertainty far away from sampled points, making it impossible to become confident in the true underlying function over the entire domain. Error-based exploration must thus be constrained to a closed region.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

19.3 Lower Confidence Bound Exploration

While error-based exploration reduces the uncertainty in the objective function overall, its samples are often in regions that are unlikely to contain a global minimum. *Lower confidence bound exploration* trades off between greedy minimization employed by prediction-based optimization and uncertainty reduction employed by error-based exploration. The next sample minimizes the *lower confidence bound* of the objective function

$$LB(\mathbf{x}) = \hat{\mu}(\mathbf{x}) - \alpha\hat{\sigma}(\mathbf{x}) \tag{19.3}$$

where $\alpha \geq 0$ is a constant that controls the trade-off between *exploration* and *exploitation*. Exploration involves minimizing uncertainty, and exploitation involves minimizing the predicted mean. We have prediction-based optimization with $\alpha = 0$, and we have error-based exploration as α approaches ∞. The process is illustrated in figure 19.3.

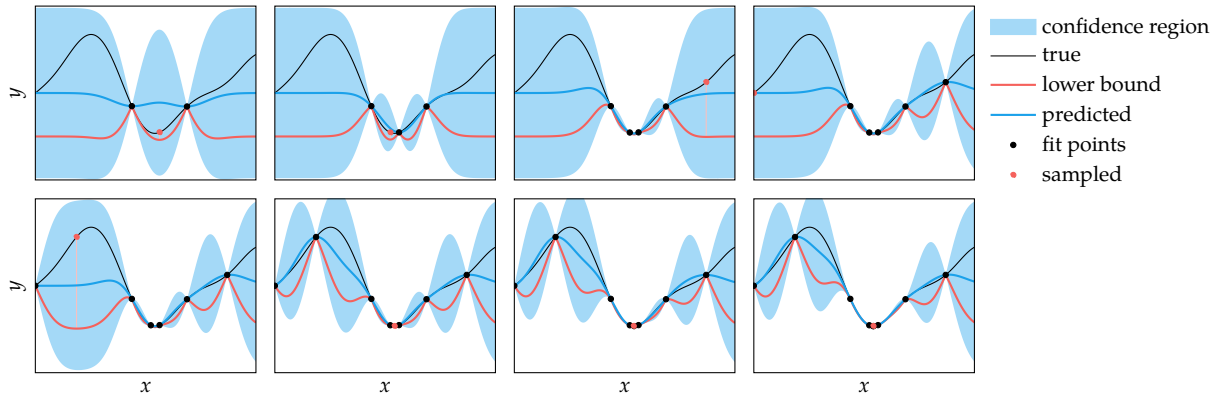


Figure 19.3. Lower confidence bound exploration trades off between minimizing uncertainty and minimizing the predicted function.

19.4 Probability of Improvement Exploration

We can sometimes obtain faster convergence by selecting the design point that maximizes the chance that the new point will be better than the samples we have seen so far. The *improvement* for a function sampled at $\mathbf{x}$ producing $y = f(\mathbf{x})$ is

$$I(y) = \begin{cases} y_{\text{min}} - y & \text{if } y < y_{\text{min}} \\ 0 & \text{otherwise} \end{cases} \tag{19.4}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

where y_{min} is the minimum value sampled so far.

The *probability of improvement* at points where $\hat{\sigma} > 0$ is

$$P(y < y_{\text{min}}) = \int_{-\infty}^{y_{\text{min}}} \mathcal{N}(y \mid \hat{\mu}, \hat{\sigma}^2) dy \tag{19.5}$$

$$= \Phi\left(\frac{y_{\text{min}} - \hat{\mu}}{\hat{\sigma}}\right) \tag{19.6}$$

where Φ is the *standard normal cumulative distribution function* (see appendix C.9). This calculation (algorithm 19.1) is shown in figure 19.4. Figure 19.5 illustrates this process. When $\hat{\sigma} = 0$, which occurs at points where we have noiseless measurements, the probability of improvement is zero.

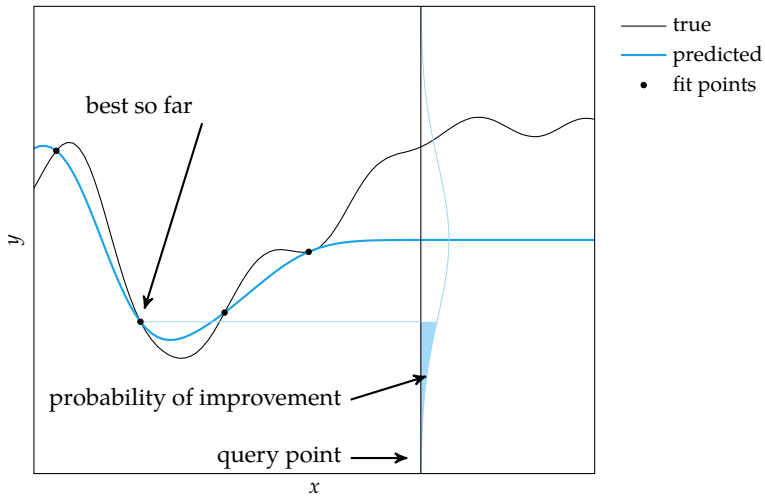


Figure 19.4. The probability of improvement is the probability that evaluating a particular point will yield a better result than the best so far. This figure shows the probability density function predicted at a query point, with the shaded region below y_{min} corresponding to the probability of improvement.

```
prob_of_improvement(y_min, μ, σ) = cdf(Normal(μ, σ), y_min)
```

Algorithm 19.1. Computing the probability of improvement for a given best y value `y_min`, mean `μ`, and standard deviation `σ`.

19.5 Expected Improvement Exploration

Optimization is concerned with finding the minimum of the objective function. While maximizing the probability of improvement will tend to decrease the objective function over time, it does not improve very much with each iteration.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

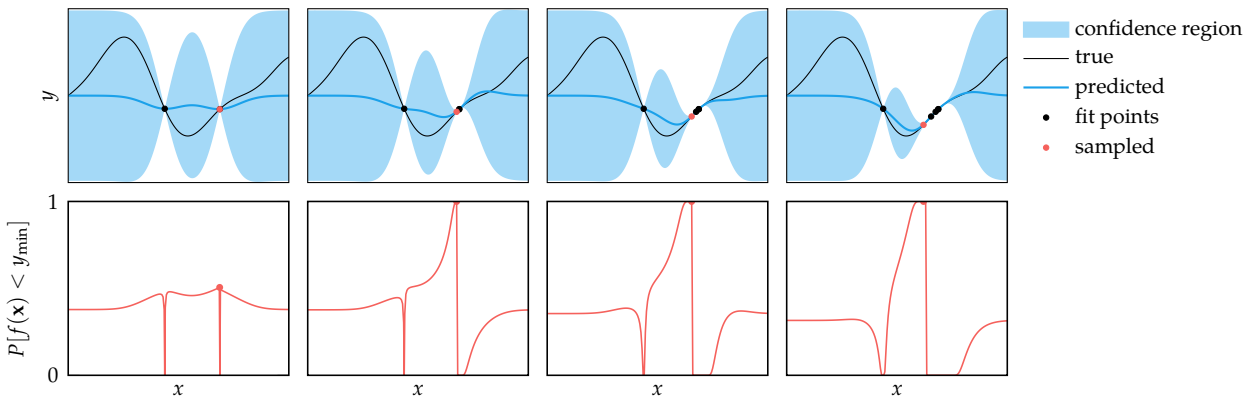


Figure 19.5. Maximizing the probability of improvement selects samples most likely to produce lower objective point values.

We can focus our exploration on points that maximize our *expected improvement* over the current best function value.

Through a substitution

$$z = \frac{y - \hat{\mu}}{\hat{\sigma}} \qquad y'_{\min} = \frac{y_{\min} - \hat{\mu}}{\hat{\sigma}} \tag{19.7}$$

we can write the improvement in equation (19.4) as

$$I(y) = \begin{cases} \hat{\sigma}(y'_{\min} - z) & \text{if } z < y'_{\min} \\ 0 & \text{otherwise} \end{cases} \tag{19.8}$$

where $\hat{\mu}$ and $\hat{\sigma}$ are the predicted mean and standard deviation at the sample point $\mathbf{x}$.

We can calculate the expected improvement using the distribution predicted by the Gaussian process:

$$\begin{aligned}
\mathbb{E}[I(y)] &= \hat{\sigma} \int_{-\infty}^{y'_{\min}} (y'_{\min} - z)\mathcal{N}(z \mid 0, 1)\, dz && (19.9)\\
&= \hat{\sigma}\left[y'_{\min} \int_{-\infty}^{y'_{\min}} \mathcal{N}(z \mid 0, 1)\, dz - \int_{-\infty}^{y'_{\min}} z\, \mathcal{N}(z \mid 0, 1)\, dz\right] && (19.10)\\
&= \hat{\sigma}\left[y'_{\min} P(z \leq y'_{\min}) + \mathcal{N}(y'_{\min} \mid 0, 1) - \underbrace{\mathcal{N}(-\infty \mid 0, 1)}_{=0}\right] && (19.11)\\
&= (y_{\min} - \hat{\mu})P(y \leq y_{\min}) + \hat{\sigma}^2 \mathcal{N}(y_{\min} \mid \hat{\mu}, \hat{\sigma}^2) && (19.12)
\end{aligned}$$

Figure 19.6 illustrates this process using algorithm 19.2.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function expected_improvement(y_min, μ, σ)
    p_imp = prob_of_improvement(y_min, μ, σ)
    p_ymin = pdf(Normal(μ, σ), y_min)
    return (y_min - μ)*p_imp + σ^2*p_ymin
end
```

Algorithm 19.2. Computing the expected improvement for a given best y value `y_min`, mean `μ`, and standard deviation `σ`.

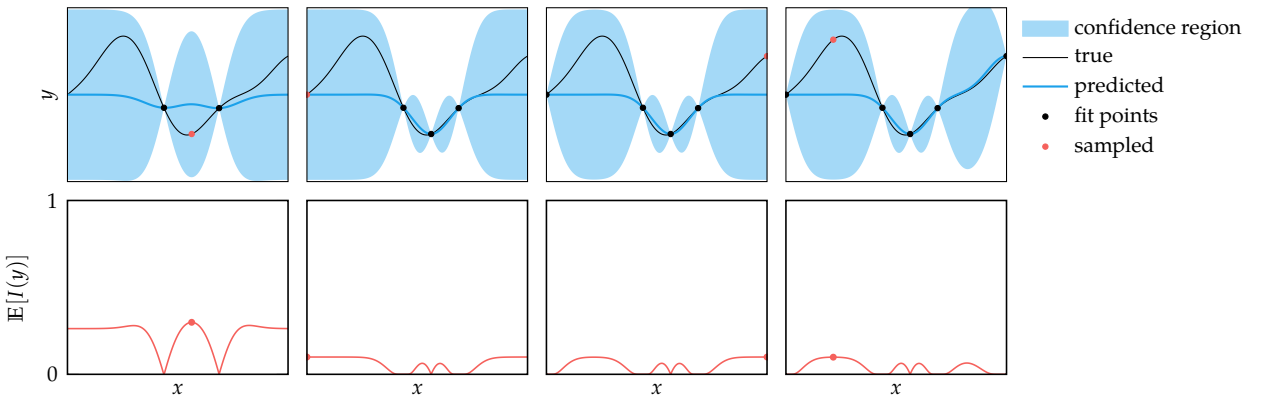


Figure 19.6. Maximizing the expected improvement selects samples that are likely to improve the best function value observed so far by as much as possible.

19.6 Safe Optimization

In some contexts, it may be costly to evaluate points that are deemed unsafe, which may correspond to low-performing or infeasible points. Problems such as the in-flight tuning of the controller of a drone or safe movie recommendations require *safe exploration*—searching for an optimal design point while carefully avoiding sampling an unsafe design.

This section outlines the *SafeOpt* algorithm,[3] which addresses a class of safe exploration problems. We sample a series of design points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$ in pursuit of a minimum but without $f(\mathbf{x}^{(i)})$ exceeding a critical safety threshold $y_{\max}$. In addition, we receive only noisy measurements of the objective function, where the noise is zero-mean with variance ν. Such an objective function and its associated safe regions are shown in figure 19.7.

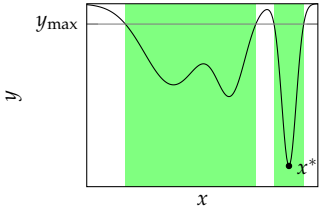


Figure 19.7. SafeOpt solves safe exploration problems that minimize f while remaining within safe regions defined by maximum objective function values.

The SafeOpt algorithm uses Gaussian process surrogate models for prediction. At each iteration, we fit a Gaussian process to the noisy samples from f. After the

[3] Y. Sui, A. Gotovos, J. Burdick, and A. Krause, "Safe Exploration for Optimization with Gaussian Processes," in *International Conference on Machine Learning (ICML)*, vol. 37, 2015.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

ith sample, SafeOpt calculates the upper and lower confidence bounds:

$$u_i(\mathbf{x}) = \hat{\mu}_{i-1}(\mathbf{x}) + \sqrt{\beta \hat{v}_{i-1}(\mathbf{x})} \tag{19.13}$$

$$\ell_i(\mathbf{x}) = \hat{\mu}_{i-1}(\mathbf{x}) - \sqrt{\beta \hat{v}_{i-1}(\mathbf{x})} \tag{19.14}$$

where larger values of β yield wider confidence regions. Such bounds are shown in figure 19.8.

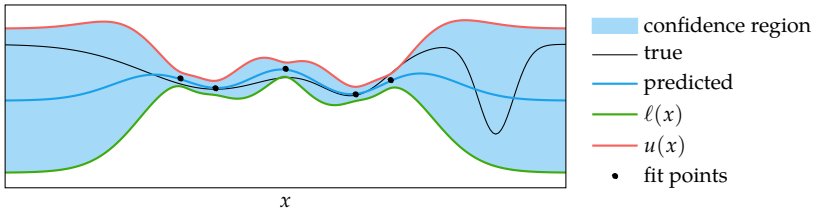


Figure 19.8. An illustration of functions based on the predictions of a Gaussian process used by SafeOpt.

The Gaussian process predicts a distribution over $f(\mathbf{x})$ for any design point. Being Gaussian, these predictions can provide only a probabilistic guarantee of safety up to an arbitrary factor:[4]

$$P(f(\mathbf{x}) \leq y_{\text{max}}) = \Phi\left(\frac{y_{\text{max}} - \hat{\mu}(\mathbf{x})}{\sqrt{\hat{v}(\mathbf{x})}}\right) \geq P_{\text{safe}} \tag{19.15}$$

[4] Note the similarity to the probability of improvement.

The predicted safe region $\mathcal{S}$ consists of the design points that provide a probability of safety greater than the required level P_{safe}, as illustrated in figure 19.9. The safe region can also be defined in terms of Lipschitz upper bounds constructed from upper bounds evaluated at previously sampled points.

SafeOpt chooses a safe sample point that balances the desire to localize a reachable minimizer of f and to expand the safe region. The set of potential minimizers of f is denoted $\mathcal{M}$ (figure 19.10), and the set of points that will potentially lead to the expansion of the safe regions is denoted $\mathcal{E}$ (figure 19.11). To trade off exploration and exploitation, we choose the design point $\mathbf{x}$ with the largest predictive variance among both sets $\mathcal{M}$ and $\mathcal{E}$.[5]

The set of potential minimizers consists of the safe points whose lower confidence bound is lower than the lowest upper bound:

$$\mathcal{M}_i = \left\{\mathbf{x} \in \mathcal{S}_i \mid \ell_i(\mathbf{x}) \leq \min_{\mathbf{x}' \in \mathcal{S}_i} u_i(\mathbf{x}')\right\} \tag{19.16}$$

[5] For a variation of this algorithm, see F. Berkenkamp, A. P. Schoellig, and A. Krause, "Safe Controller Optimization for Quadrotors with Gaussian Processes," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

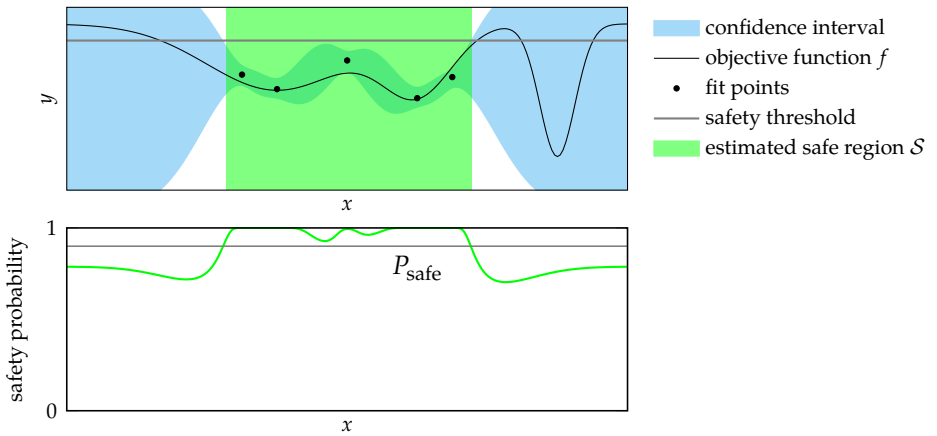


Figure 19.9. The safety regions (green) predicted by a Gaussian process.

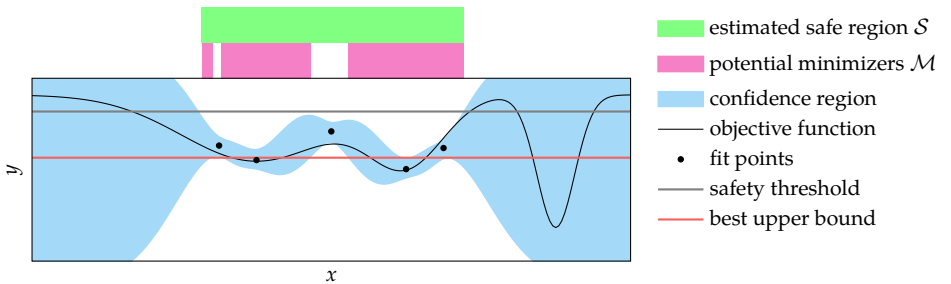


Figure 19.10. The potential minimizers are the safe points whose lower bounds are lower than the best, safe upper bound.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

At step i, the set of potential expanders $\mathcal{E}_i$ consists of the safe points that, if added to the Gaussian process, optimistically assuming the lower bound, produce a posterior distribution with a larger safe set. The potential expanders naturally lie near the boundary of the safe region.

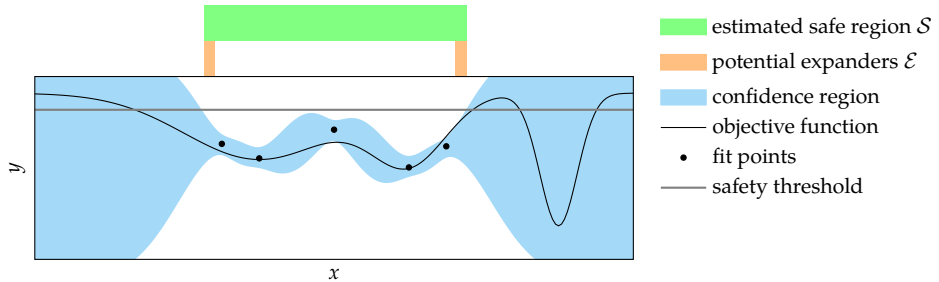


Figure 19.11. The set of potential expanders.

Given an initial safe point[6] $\mathbf{x}^{(1)}$, SafeOpt chooses the design point among sets $\mathcal{M}$ and $\mathcal{E}$ with the greatest uncertainty, as quantified by the width $w_i(x) = u(x) - \ell(x)$:

$$x^{(i)} = \underset{\mathbf{x} \in \mathcal{M}_i \cup \mathcal{E}_i}{\arg\max}\, w_i(\mathbf{x}) \tag{19.17}$$

[6] SafeOpt cannot guarantee safety if it is not initialized with at least one point that it knows is safe.

SafeOpt proceeds until a termination condition is met. It is common to run the algorithm for a fixed number of iterations or until the maximum width is less than a set threshold.

Maintaining sets in multidimensional spaces can be computationally challenging. SafeOpt assumes a finite design space $\mathcal{X}$ that can be obtained with a sampling method applied over the continuous search domain. Increasing the density of the finite design space leads to more accurate results with respect to the continuous space, but it takes longer per iteration.

SafeOpt is implemented in algorithm 19.3, and calls algorithm 19.4 to update the predicted confidence intervals; algorithm 19.5 to compute the safe, minimizer, and expander regions; and algorithm 19.6 to select a query point. The progression of SafeOpt is shown for one dimension in figure 19.12, and for two dimensions in figure 19.13.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function safe_opt(GP, X, i, f, y_max; β=3.0, k_max=10)
    push!(GP, X[i], f(X[i])) # make first observation

    m = length(X)
    u, l = fill(Inf, m), fill(-Inf, m)
    S, M, E = falses(m), falses(m), falses(m)

    for k in 1 : k_max
        update_confidence_intervals!(GP, X, u, l, β)
        compute_sets!(GP, S, M, E, X, u, l, y_max, β)
        i = safeopt_query_point(M, E, u, l)
        i != 0 || break
        push!(GP, X[i], f(X[i]))
    end

    # return the best point
    update_confidence_intervals!(GP, X, u, l, β)
    S .= u .≤ y_max
    if any(S)
        u_best, i_best = findmin(u[S])
        i_best = findfirst(isequal(i_best), cumsum(S))
        return (u_best, i_best)
    else
        return (NaN,0)
    end
end
```

Algorithm 19.3. The SafeOpt algorithm applied to an empty Gaussian process `GP`, a finite design space `X`, index of initial safe point `i`, objective function `f`, and safety threshold `y_max`. The optional parameters are the confidence scalar `β` and the number of iterations `k_max`. A tuple containing the best safe upper bound and its index in `X` is returned.

```
function update_confidence_intervals!(GP, X, u, l, β)
    μₚ, νₚ = predict(GP, X)
    u .= μₚ + sqrt.(β*νₚ)
    l .= μₚ - sqrt.(β*νₚ)
    return (u, l)
end
```

Algorithm 19.4. A method for updating the lower and upper bounds used in SafeOpt, which takes the Gaussian process `GP`, the finite search space `X`, the upper and lower-bound vectors `u` and `l`, and the confidence scalar `β`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function compute_sets!(GP, S, M, E, X, u, l, y_max, β)
    fill!(M, false)
    fill!(E, false)

    # safe set
    S .= u .≤ y_max

    if any(S)

        # potential minimizers
        M[S] = l[S] .< minimum(u[S])

        # maximum width (in M)
        w_max = maximum(u[M] - l[M])

        # expanders - skip values in M or those with w ≤ w_max
        E .= S .& .~M # skip points in M
        if any(E)
            E[E] .= maximum(u[E] - l[E]) .> w_max
            for (i,e) in enumerate(E)
                if e && u[i] - l[i] > w_max
                    push!(GP, X[i], l[i])
                    μₚ, νₚ = predict(GP, X[.~S])
                    pop!(GP)
                    E[i] = any(μₚ + sqrt.(β*νₚ) .≤ y_max)
                    if E[i]; w_max = u[i] - l[i]; end
                end
            end
        end
    end

    return (S,M,E)
end
```

Algorithm 19.5. A method for updating the safe `S`, minimizer `M`, and expander `E` sets used in SafeOpt. The sets are all Boolean vectors indicating whether the corresponding design point in `X` is in the set. The method also takes the Gaussian process `GP`, the upper and lower bounds `u` and `l`, respectively, the safety threshold `y_max`, and the confidence scalar `β`.

```
function safeopt_query_point(M, E, u, l)
    ME = M .| E
    if any(ME)
        v = argmax(u[ME] - l[ME])
        return findfirst(isequal(v), cumsum(ME))
    else
        return 0
    end
end
```

Algorithm 19.6. A method for obtaining the next query point in SafeOpt. The index of the point in `X` with the greatest width is returned.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

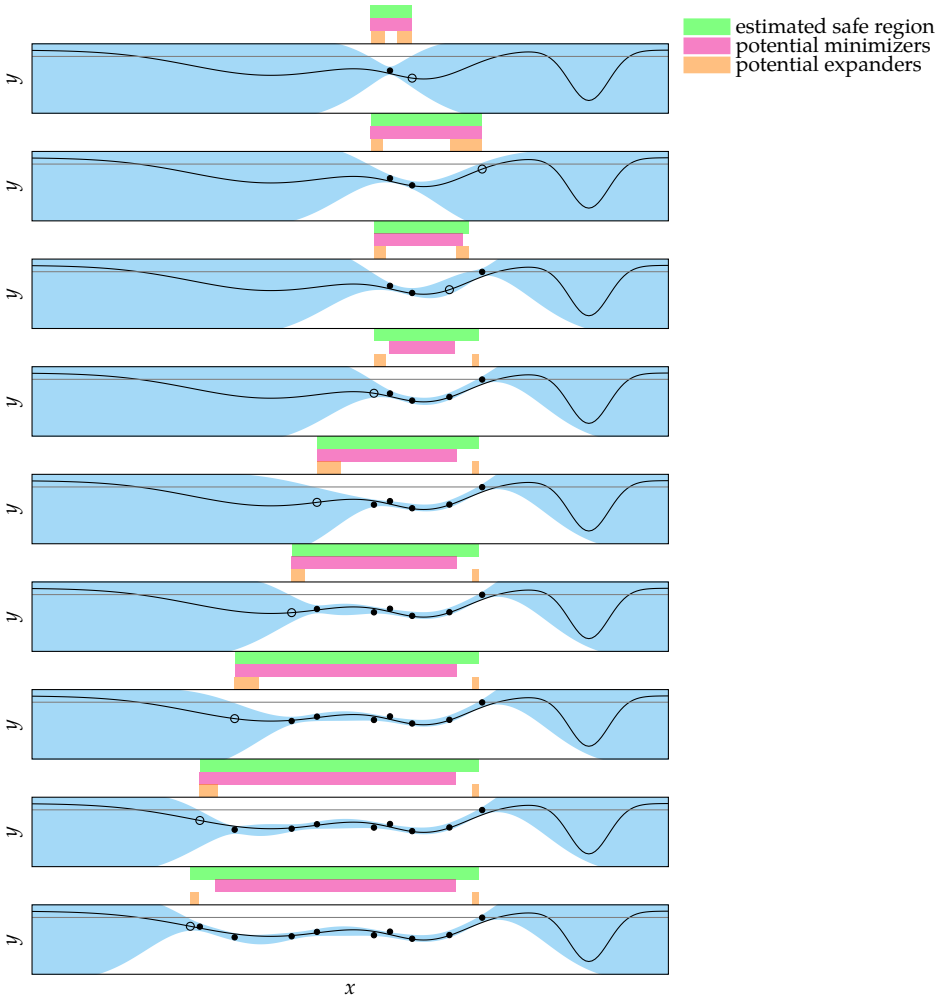


Figure 19.12. The first nine iterations of SafeOpt on a univariate function. SafeOpt can never reach the global optimum on the right-hand side because it requires crossing an unsafe region. We can only hope to find the global minima in our locally reachable safe region.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

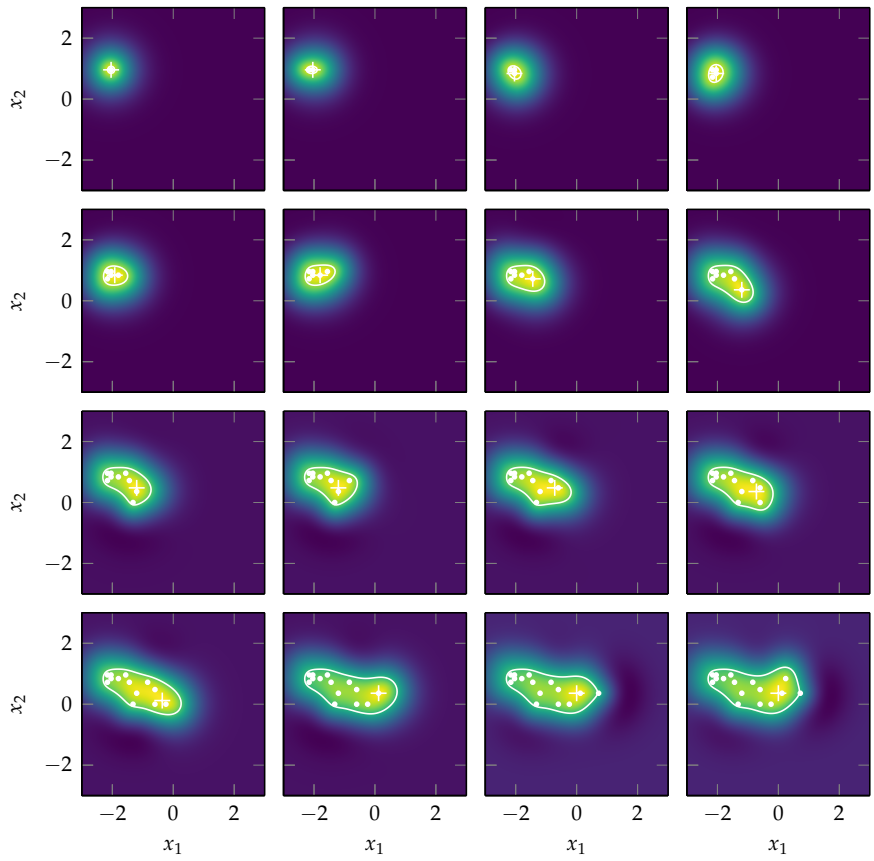
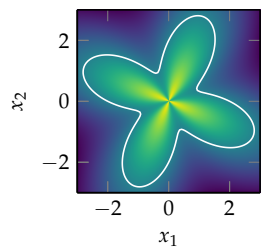


Figure 19.13. SafeOpt applied to the flower function (appendix B.4) with $y_{\max} = 2$, a Gaussian process mean of $\mu(\mathbf{x}) = 2.5$, variance $\nu = 0.01, \beta = 10$, a 51×51 uniform grid over the search space, and an initial point $x^{(1)} = [-2.04, 0.96]$. The color indicates the value of the upper bound, the cross indicates the safe point with the lowest upper bound, and the white contour line is the estimated safe region.

The objective function with the true safe region outlined in white:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

19.7 Summary

- Gaussian processes can be used to guide the optimization process using a variety of strategies that use estimates of quantities such as the lower confidence bound, probability of improvement, and expected improvement.
- Some problems do not allow for the evaluation of unsafe designs, in which case we can use safe exploration strategies that rely on Gaussian processes.

19.8 Exercises

Exercise 19.1. Give an example in which prediction-based optimization fails.

Solution: Prediction-based optimization with Gaussian processes can repeatedly sample the same point. Suppose we have a zero-mean Gaussian process and we start with a single point $\mathbf{x}^{(1)}$, which gives us some $y^{(1)}$. The predicted mean has a single global minimizer at $\mathbf{x}^{(1)}$. Prediction-based optimization will continue to sample at $\mathbf{x}^{(1)}$.

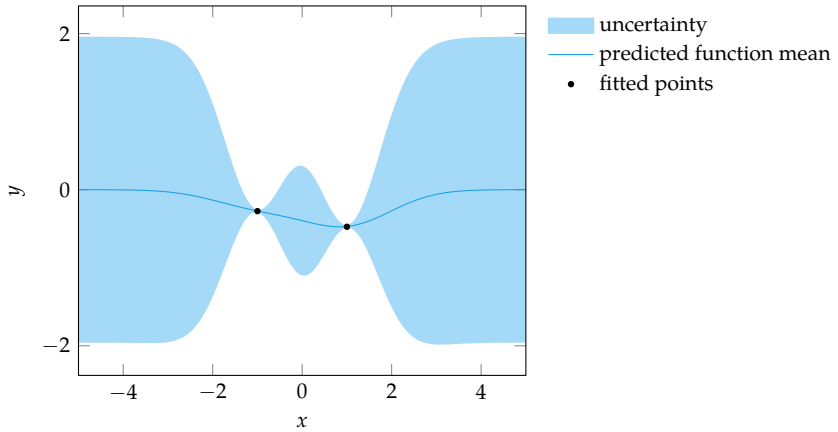
Exercise 19.2. What is the main difference between lower confidence bound exploration and error-based exploration in the context of optimization?

Solution: Error-based exploration wastes effort in reducing the variance and does not actively seek to minimize the function.

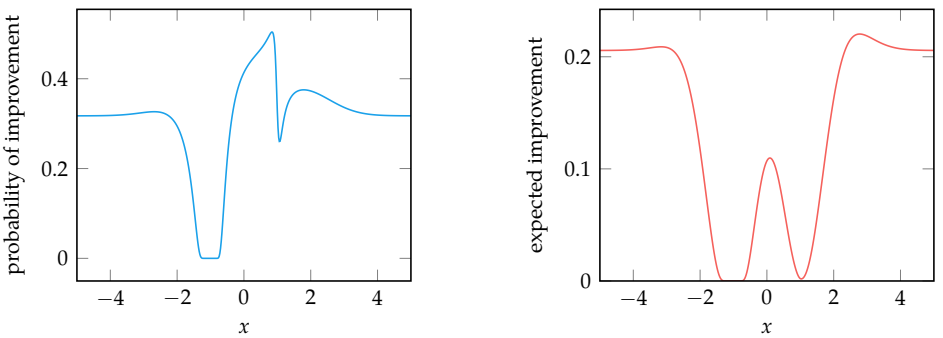
Exercise 19.3. We have a function $f(x) = (x-2)^2/40 - 0.5$ with $x \in [-5, 5]$, and we have evaluation points at -1 and 1. Assume we use a Gaussian process surrogate model with a zero-mean function, and a squared exponential kernel $\exp(-r^2/2)$, where r is the Euclidean distance between two points. Which value for x would we evaluate next if we were maximizing probability of improvement? Which value for x would we evaluate next if we were maximizing expected improvement?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The Gaussian process looks like this:



The probability of improvement and expected improvement look like:



The maximum probability of improvement is at $x = 0.84$ for $P = 0.504$.
The maximum expected improvement is at $x = 2.78$ for $E = 0.22$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

20 Optimization Under Uncertainty

Previous chapters assumed that the optimization objective is to minimize a deterministic function of our design points. In many engineering tasks, however, there may be uncertainty in the objective function or the constraints. Uncertainty may arise due to a number of factors, such as model approximations, imprecision, and fluctuations of parameters over time. This chapter covers a variety of methods for accounting for uncertainty in our optimization to enhance robustness.[1]

[1] Additional references include: H.-G. Beyer and B. Sendhoff, "Robust Optimization—A Comprehensive Survey," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 33, pp. 3190–3218, 2007. G.-J. Park, T.-H. Lee, K. H. Lee, and K.-H. Hwang, "Robust Design: An Overview," *AIAA Journal*, vol. 44, no. 1, pp. 181–191, 2006.

20.1 Uncertainty

Uncertainty in the optimization process can arise for a variety of reasons. There may be *irreducible uncertainty*,[2] which is inherent to the system, such as background noise, varying material properties, and quantum effects. These uncertainties cannot be avoided and our design should accommodate them. There may also be *epistemic uncertainty*,[3] which is uncertainty caused by a subjective lack of knowledge by the designer. This uncertainty can arise from approximations in the model[4] used when formulating the design problem and errors introduced by numerical solution methods.

[2] This form of uncertainty is sometimes called *aleatory uncertainty* or *random uncertainty*.

[3] Epistemic uncertainty is also called *reducible uncertainty*.

[4] The statistician George Box famously wrote: *All models are wrong; some models are useful.* G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, 2nd ed. Wiley, 2005. p. 440.

Accounting for these various forms of uncertainty is critical to ensuring robust designs. In this chapter, we will use $\mathbf{z} \in \mathcal{Z}$ to represent a vector of random values. We want to minimize $f(\mathbf{x}, \mathbf{z})$, but we do not have control over $\mathbf{z}$. Feasibility depends on both the design vector $\mathbf{x}$ and the uncertain vector $\mathbf{z}$. This chapter introduces the feasible set over $\mathbf{x}$ and $\mathbf{z}$ pairs as $\mathcal{F}$. We have feasibility if and only if $(\mathbf{x}, \mathbf{z}) \in \mathcal{F}$. We will use $\mathcal{X}$ as the design space, which may include potentially infeasible designs depending on the value of $\mathbf{z}$.

Optimization with uncertainty was briefly introduced in section 18.5 in the context of using a Gaussian process to represent an objective function inferred

from noisy measurements. We had $f(\mathbf{x}, z) = f(\mathbf{x}) + z$ with the additional assumption that z comes from a zero-mean Gaussian distribution.[5] Uncertainty may be incorporated into the evaluation of a design point in other ways. For example, if we had noise in the input to the objective function,[6] we might have $f(\mathbf{x}, \mathbf{z}) = f(\mathbf{x} + \mathbf{z})$. In general, $f(\mathbf{x}, \mathbf{z})$ can be a complex, nonlinear function of $\mathbf{x}$ and $\mathbf{z}$. In addition, $\mathbf{z}$ may not come from a Gaussian distribution; in fact, it may come from a distribution that is not known.

[5] Here, the two-argument version of f takes as input the design point and random vector, but the single-argument version of f represents a deterministic function of the design point without noise.

[6] For example, there may be variability in the manufacturing of our design.

Figure 20.1 demonstrates how the degree of uncertainty can affect our choice of design. For simplicity, x is a scalar and z is selected from a zero-mean Gaussian distribution. We assume that z corresponds to noise in the input to f, and so $f(x, z) = f(x + z)$. The figure shows the expected value of the objective function for different levels of noise. The global minimum without noise is a. However, aiming for a design near a can be risky since it lies within a steep valley, making it rather sensitive to noise. Even with low noise, it may be better to choose a design near b. Designs near c can provide even greater robustness to larger amounts of noise. If the noise is very high, the best design might even fall between b and c, which corresponds to a local maximum in the absence of noise.

There are a variety of different ways to account for uncertainty in optimization. We will discuss both set-based uncertainty and probabilistic uncertainty.[7]

[7] Other approaches for representing uncertainty include *Dempster-Shafer theory*, *fuzzy-set theory*, and *possibility theory*, which are beyond the scope of this book.

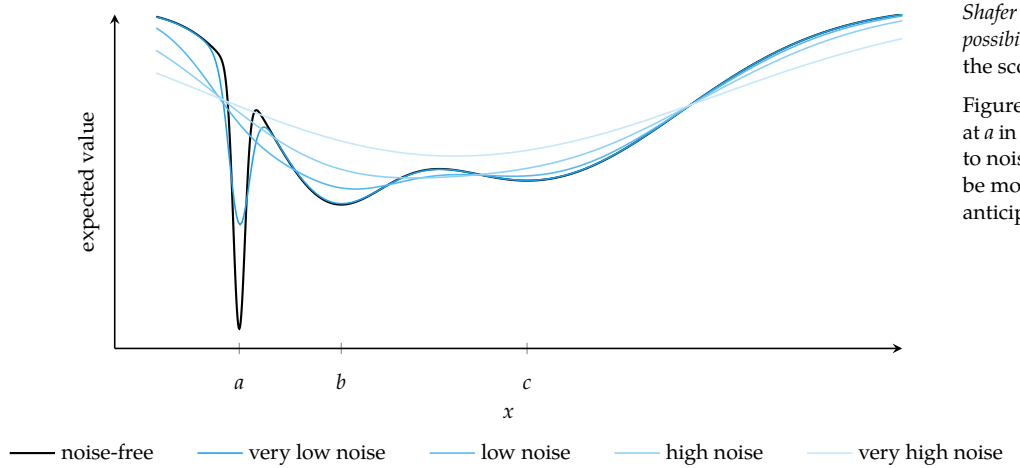


Figure 20.1. The global minimum at a in the noiseless case is sensitive to noise. Other design points may be more robust depending on the anticipated level of noise.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

20.2 Set-Based Uncertainty

Set-based uncertainty approaches assume that $\mathbf{z}$ belongs to a set $\mathcal{Z}$, but these approaches make no assumptions about the relative likelihood of different points within that set. The set $\mathcal{Z}$ can be defined in different ways. One way is to define intervals for each component of $\mathcal{Z}$. Another way is to define $\mathcal{Z}$ by a set of inequality constraints, $\mathbf{g}(\mathbf{x}, \mathbf{z}) \leq \mathbf{0}$, similar to what was done for the design space $\mathcal{X}$ in chapter 10. In general, we can have $\mathcal{Z}$ depend on $\mathbf{x}$, but we will omit this potential dependency in our discussion for simplicity.

20.2.1 Minimax

In problems with set-based uncertainty, we often want to minimize the maximum possible value of the objective function. Such a *minimax* approach[8] solves the optimization problem

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}} \underset{\mathbf{z}\in\mathcal{Z}}{\text{maximize}}\, f(\mathbf{x}, \mathbf{z}) \tag{20.1}$$

[8] Also called the *robust counterpart approach* or *robust regularization*.

In other words, we want to find an $\mathbf{x}$ that minimizes f, assuming the worst-case value for $\mathbf{z}$.

This optimization is equivalent to defining a modified objective function[9]

$$f_{\text{mod}}(\mathbf{x}) = \underset{\mathbf{z}\in\mathcal{Z}}{\text{maximize}}\, f(\mathbf{x}, \mathbf{z}) \tag{20.2}$$

and then solving

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\, f_{\text{mod}}(\mathbf{x}) \tag{20.3}$$

[9] The minimax problem preserves convexity. In other words, if $f(\mathbf{x}, \mathbf{z})$ is convex in $\mathbf{x}$ for each fixed $\mathbf{z}$, then $f_{\text{mod}}(\mathbf{x}) = \text{maximize}_{\mathbf{z}\in\mathcal{Z}}\, f(\mathbf{x}, \mathbf{z})$ is also convex in $\mathbf{x}$, allowing us to use convex optimization methods to solve the minimax problem.

Example 20.1 shows this optimization on a univariate problem and illustrates the effect of different levels of uncertainty.

In problems where we have feasibility constraints, our optimization problem becomes

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}} \underset{\mathbf{z}\in\mathcal{Z}}{\text{maximize}}\, f(\mathbf{x}, \mathbf{z}) \text{ subject to } (\mathbf{x}, \mathbf{z}') \in \mathcal{F} \text{ for all } \mathbf{z}' \in \mathcal{Z} \tag{20.4}$$

Example 20.2 shows the effect of applying minimax on the space of feasible design points when there are constraints.

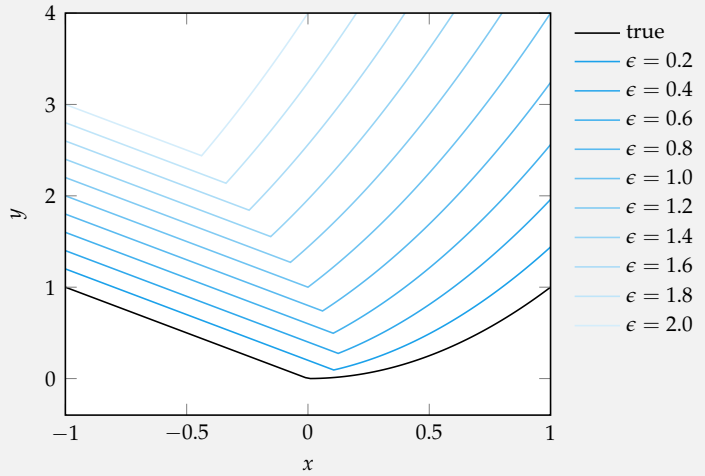
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 20.1. Example of a minimax approach to optimization under set-based uncertainty.

Consider the objective function

$$f(x,z) = f(x+z) = f(\tilde{x}) = \begin{cases} -\tilde{x} & \text{if } \tilde{x} \leq 0 \\ \tilde{x}^2 & \text{otherwise} \end{cases}$$

where $\tilde{x} = x + z$, with a set-based uncertainty region $z \in [-\epsilon, \epsilon]$. The minimax approach is a minimization problem over the modified objective function $f_{\text{mod}}(x) = \text{maximize}_{z \in [-\epsilon,\epsilon]} f(x,z)$.



The figure above shows $f_{\text{mod}}(x)$ for several different values of ϵ. The minimum for $\epsilon = 0$ coincides with the minimum of $f(x,0)$. As ϵ is increased, the minimum first shifts right as x increases faster than x^2 and then shifts left as x^2 increases faster than x. The robust minimizer does not generally coincide with the minimum of $f(x,0)$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

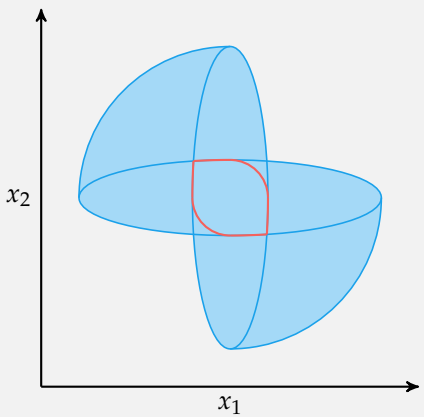
Example 20.2. The minimax approach applied to uncertainty in the feasible set.

Consider an uncertain feasible set in the form of a rotated ellipse, where $(\mathbf{x}, z) \in \mathcal{F}$ if and only if $z \in [0, \pi/2]$ and

$$(x_1 \cos z + x_2 \sin z)^2 + (x_1 \sin z - x_2 \cos z)^2 / 16 \leq 1$$

When $z = 0$, the major axis of the ellipse is vertical. Increasing the value of z rotates it counterclockwise to horizontal at $z = \pi/2$. The figure below shows the vertical and horizontal ellipses and the set of all points that are feasible for at least one z in blue.

A minimax approach to optimization should consider only design points that are feasible under all values of z. The set of designs that are always feasible is given by the intersection of all ellipses formed by varying z. This set is outlined in red.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

20.2.2 *Information-Gap Decision Theory*

Instead of assuming the uncertainty set $\mathcal{Z}$ is fixed, an alternative approach known as *information-gap decision theory*[10] parameterizes the uncertainty set by a nonnegative scalar *gap* parameter ϵ. The gap controls the volume of the parameterized set $\mathcal{Z}(\epsilon)$ centered at some nominal value $\bar{\mathbf{z}} = \mathcal{Z}(0)$. One way to define $\mathcal{Z}(\epsilon)$ is as a hypersphere of radius ϵ centered at a nominal point $\bar{\mathbf{z}}$:

$$\mathcal{Z}(\epsilon) = \{\mathbf{z} \mid \|\mathbf{z} - \bar{\mathbf{z}}\|_2 \leq \epsilon\} \tag{20.5}$$

Figure 20.2 illustrates this definition in two dimensions.

By parameterizing the uncertainty set, we avoid committing to a particular uncertainty set. Uncertainty sets that are too large sacrifice the quality of the solution, and uncertainty sets that are too small sacrifice robustness. Design points that remain feasible for larger gaps are considered more robust.

In information-gap decision theory, we try to find the design point that allows for the largest gap while preserving feasibility. This design point can be obtained by solving the following optimization problem:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathcal{X}} \operatorname*{maximize}_{\epsilon \in [0, \infty)} \begin{cases} \epsilon & \text{if } (\mathbf{x}, \mathbf{z}) \in \mathcal{F} \text{ for all } \mathbf{z} \in \mathcal{Z}(\epsilon) \\ 0 & \text{otherwise} \end{cases} \tag{20.6}$$

This optimization focuses on finding designs that ensure feasibility in the presence of uncertainty. In fact, equation (20.6) does not explicitly include the objective function f. However, we can incorporate the constraint that $f(\mathbf{x}, \mathbf{z})$ be no greater than some threshold $y_{\max}$. Such performance constraints can help us avoid excessive risk aversion. Figure 20.3 and example 20.3 illustrate the application of information-gap decision theory.

[10] F. M. Hemez and Y. Ben-Haim, "Info-Gap Robustness for the Correlation of Tests and Simulations of a Non-Linear Transient," *Mechanical Systems and Signal Processing*, vol. 18, no. 6, pp. 1443–1467, 2004.

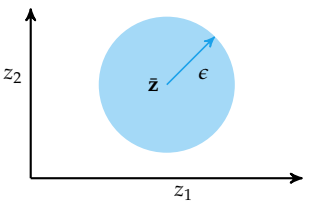


Figure 20.2. A parameterized uncertainty set $\mathcal{Z}(\epsilon)$ in the form of a hypersphere.

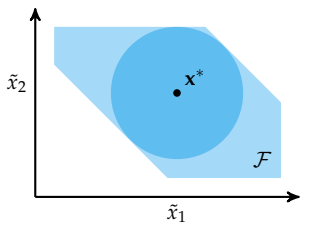


Figure 20.3. Information-gap decision theory applied to an objective function with additive noise $f(\tilde{\mathbf{x}})$ with $\tilde{\mathbf{x}} = \mathbf{x} + \mathbf{z}$ and a circular uncertainty set

$$\mathcal{Z}(\epsilon) = \{\mathbf{z} \mid \|z\|_2 \leq \epsilon\}$$

The design $\mathbf{x}^*$ is optimal under information-gap decision theory as it allows for the largest possible ϵ such that all $\mathbf{x}^* + \mathbf{z}$ are feasible.

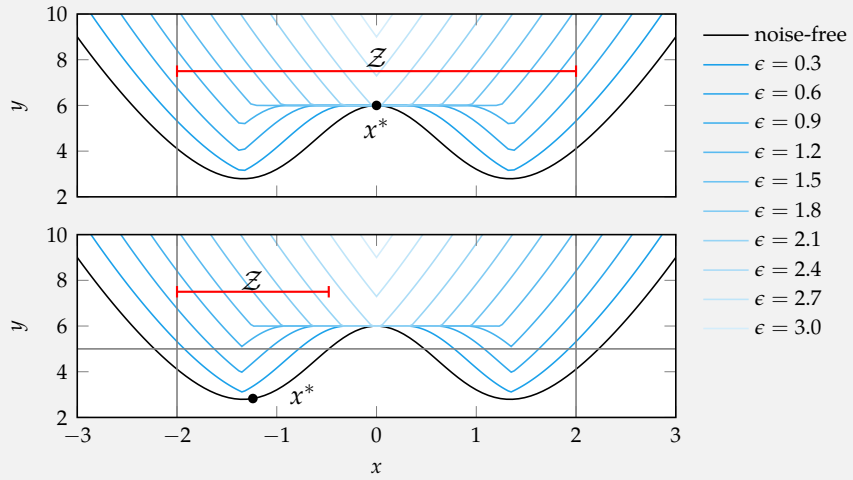
20.3 *Probabilistic Uncertainty*

Models of *probabilistic uncertainty* use distributions over a set $\mathcal{Z}$. Probabilistic uncertainty models provide more information than set-based uncertainty models, allowing the designer to account for the probability of different outcomes of a design. These distributions can be defined using expert knowledge or learned from data. Given a distribution p over $\mathcal{Z}$, we can infer a distribution over the output of f using methods that will be discussed in chapter 21. In general, the

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 20.3. We can mitigate excessive risk aversion by applying a constraint on the maximum acceptable objective function value when applying information-gap decision theory.

Consider the robust optimization of $f(x,z) = \tilde{x}^2 + 6e^{-\tilde{x}^2}$ with $\tilde{x} = x + z$ subject to the constraint $\tilde{x} \in [-2,2]$ with the uncertainty set $\mathcal{Z}(\epsilon) = [-\epsilon,\epsilon]$.



Applying information-gap decision theory to this problem results in a maximally sized uncertainty set and a design centered in the suboptimal region of the objective function. Applying an additional constraint on the maximum objective function value, $f(x,z) \leq 5$, allows the same approach to find a design with better noise-free performance. The blue lines indicate the worst-case objective function value for a given uncertainty parameter ϵ.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

distribution p may depend on $\mathbf{x}$, but we will omit this dependency for simplicity. This section outlines five different metrics for converting this distribution into a scalar value given a particular design $\mathbf{x}$. We can then optimize with respect to these metrics.[11]

We can also combine elements of the previous section on set-based uncertainty with probabilistic uncertainty. For example, we may be uncertain about the distribution p, in which case we can consider a set of possible probability distributions $\mathcal{P}$. In this case, we can optimize over the worst-case distribution in $\mathcal{P}$. This approach is known as *distributionally robust optimization*.[12]

[11] Further discussion of various metrics can be found in A. Shapiro, D. Dentcheva, and A. Ruszczyński, *Lectures on Stochastic Programming: Modeling and Theory*, 2nd ed. SIAM, 2014.

[12] A. Shapiro, "Distributionally Robust Stochastic Programming," *SIAM Journal on Optimization*, vol. 27, no. 4, pp. 2258–2275, 2017.

20.3.1 Expected Value

One way to convert the distribution output by f into a scalar value is to use the *expected value* or *mean*. The expected value is the average output that we can expect when considering all outputs of $f(\mathbf{x}, \mathbf{z})$ for all $\mathbf{z} \in \mathcal{Z}$ and their corresponding probabilities. The expected value as a function of the design point $\mathbf{x}$ is

$$\mathbb{E}_{\mathbf{z}\sim p}[f(\mathbf{x}, \mathbf{z})] = \int_{\mathcal{Z}} f(\mathbf{x}, \mathbf{z}) p(\mathbf{z})\, d\mathbf{z} \tag{20.7}$$

The expected value does not necessarily correspond to the objective function without noise, as illustrated in example 20.4.

Computing the integral in equation (20.7) analytically may not be possible. One may approximate that value using sampling or a variety of other more sophisticated techniques discussed in chapter 21.

20.3.2 Variance

Besides optimizing with respect to the expected value of the function, we may also be interested in choosing design points whose value is not overly sensitive to uncertainty.[13] Such regions can be quantified using the *variance* of f:

$$\begin{aligned} \operatorname{Var}[f(\mathbf{x}, \mathbf{z})] &= \mathbb{E}_{\mathbf{z}\sim p}\left[\left(f(\mathbf{x}, \mathbf{z}) - \mathbb{E}_{\mathbf{z}\sim p}[f(\mathbf{x}, \mathbf{z})]\right)^2\right] && (20.8) \\ &= \int_{\mathcal{Z}} f(\mathbf{x}, \mathbf{z})^2 p(\mathbf{z})\, d\mathbf{z} - \mathbb{E}_{\mathbf{z}\sim p}[f(\mathbf{x}, \mathbf{z})]^2 && (20.9) \end{aligned}$$

[13] Sometimes designers seek plateau-like regions where the output of the objective function is relatively constant, such as producing materials with consistent performance or scheduling trains such that they arrive at a consistent time.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

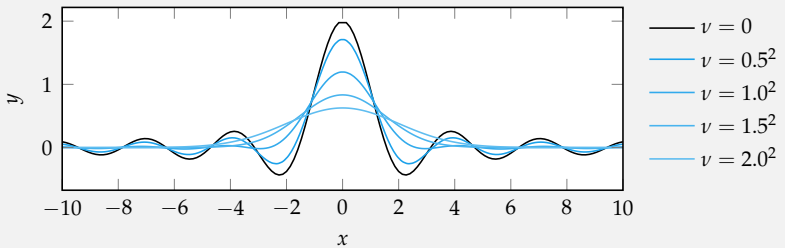
Example 20.4. The expected value of an uncertain objective function depends on how the uncertainty is incorporated into the objective function.

One common model is to apply zero-mean Gaussian noise to the function output, $f(\mathbf{x}, \mathbf{z}) = f(\mathbf{x}) + \mathbf{z}$, as was the case with Gaussian processes in chapter 18. The expected value is equivalent to the noise-free case:

$$\mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[f(\mathbf{x}) + \mathbf{z}] = \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[f(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathbf{z}] = f(\mathbf{x})$$

It is also common to add noise directly to the design vector, $f(\mathbf{x}, \mathbf{z}) = f(\mathbf{x} + \mathbf{z}) = f(\tilde{\mathbf{x}})$. In such cases the expected value is affected by the variance of zero-mean Gaussian noise.

Consider minimizing the expected value of $f(\tilde{x}) = \sin(2\tilde{x})/\tilde{x}$ with $\tilde{x} = x + z$ for z drawn from a zero-mean Gaussian distribution $\mathcal{N}(0, \nu)$. Increasing the variance increases the effect that the local function landscape has on a design.



The plot above shows that changing the variance affects the location of the optima.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We call design points with large variance *sensitive* and design points with small variance *robust*. Examples of sensitive and robust points are shown in figure 20.4. We are typically interested in good points as measured by their expected value that are also robust. Managing the trade-off between the expected objective function value and the variance is a multiobjective optimization problem (see examples 20.5 and 20.6), and we can use techniques discussed in chapter 15.

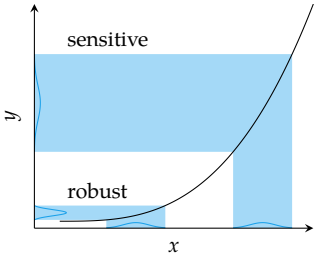


Figure 20.4. Probabilistic approaches produce probability distributions over the model output. Design points can be sensitive or robust to uncertainty. The blue regions show how the distribution over a normally distributed design is affected by the objective function.

20.3.3 Statistical Feasibility

An alternative metric against which to optimize is *statistical feasibility*. Given $p(\mathbf{z})$, we can compute the probability a design point $\mathbf{x}$ is feasible:

$$P(\mathbf{x} \in \mathcal{F}) = \int_{\mathcal{Z}} ((\mathbf{x}, \mathbf{z}) \in \mathcal{F}) p(\mathbf{z})\, d\mathbf{z} \tag{20.10}$$

This probability can be estimated through sampling. If we are also interested in ensuring that the objective value does not exceed a certain threshold, we can incorporate a constraint $f(\mathbf{x}, \mathbf{z}) \leq y_{\text{max}}$ as is done with information-gap decision theory. Unlike the expected value and variance metrics, we want to maximize this metric.

20.3.4 Value at Risk

The *value at risk* (*VaR*) is a *risk measure* corresponding to the best objective value that can be guaranteed with probability α. We can write this definition mathematically in terms of the *cumulative distribution function*, denoted $\Phi(y)$, over the random output of the objective function. The probability that the outcome is less than or equal to y is given by $\Phi(y)$. VaR with confidence α is the minimum value of y such that $\Phi(y) \geq \alpha$. This definition is equivalent to the α *quantile* of a probability distribution. An α close to 1 is sensitive to unfavorable outliers,[14] whereas an α close to 0 is overly optimistic and close to the best possible outcome.

20.3.5 Conditional Value at Risk

The *conditional value at risk* (*CVaR*) is another risk measure closely related to the value at risk.[15] CVaR is the expected value of the top $1 - \alpha$ quantile of the probability distribution over the output. This quantity is illustrated in figure 20.5.

[14] Setting $\alpha = 1$ when using VaR is equivalent to a minimax approach with set-based uncertainty where the optimization is done assuming the worst-case outcome.

[15] The conditional value at risk is also known as the *mean excess loss*, *mean shortfall*, and *tail value at risk*. R. T. Rockafellar and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, vol. 2, pp. 21–42, 2000. It is also a kind of *coherent risk measure*, which means that it satisfies some additional mathematical properties. Another coherent risk measure, not discussed here, is the *entropic value at risk*. A. Ahmadi-Javid, "Entropic Value-At-Risk: A New Coherent Risk Measure," *Journal of Optimization Theory and Applications*, vol. 155, no. 3, pp. 1105–1123, 2011.

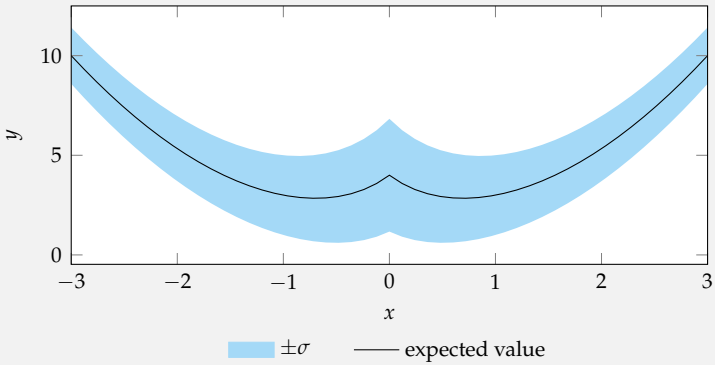
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 20.5. Considering both the expected value and the variance in optimization under uncertainty.

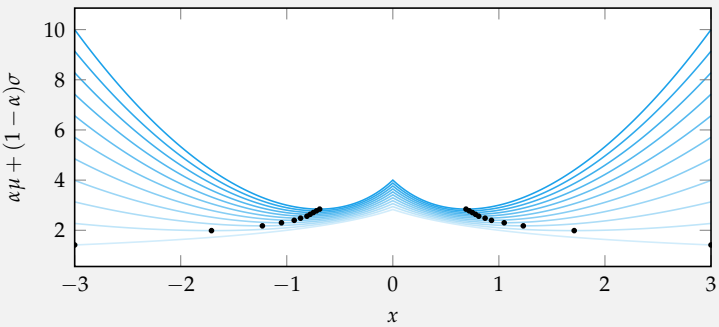
Consider the objective function $f(x, z) = x^2 + z$, with z drawn from a Gamma distribution that depends on x. We can construct a function `dist(x)` that returns a `Gamma` distribution from the `Distributions.jl` package:

```
dist(x) = Gamma(2/(1+abs(x)),2)
```

This distribution has mean $4/(1 + |x|)$ and variance $8/(1 + |x|)$.



We can find a robust optimizer that minimizes both the expected value and the variance. Minimizing with respect to the expected value, ignoring the variance, produces two minima at $x \approx \pm 0.695$. Incorporating a penalty for the variance shifts these minima away from the origin. The figure below shows objective functions of the form $\alpha\,\mathbb{E}[y \mid x] + (1 - \alpha)\sqrt{\text{Var}[y \mid x]}$ for $\alpha \in [0, 1]$ along with their associated minima.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The mean-variance trade-off often arises in finance, where there is uncertainty in return on investments. Suppose we have a budget b available for investment and a design $\mathbf{x}$ is a stock portfolio, with x_i being the amount of money allocated to the ith stock. Stock returns $\mathbf{z}$ are modeled as Gaussian with $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The return of a portfolio is then

$$y = f(\mathbf{x}, \mathbf{z}) = x_1 z_1 + x_2 z_2 + \ldots = \mathbf{x}^\top \mathbf{z}$$

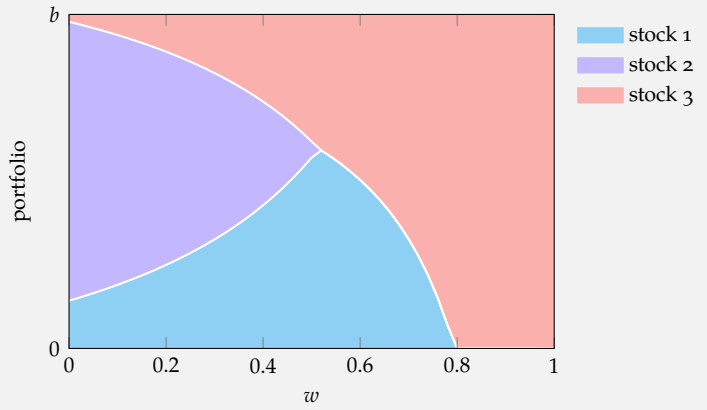
which is also Gaussian with

$$y \sim \mathcal{N}(\mathbf{x}^\top \boldsymbol{\mu}, \mathbf{x}^\top \boldsymbol{\Sigma} \mathbf{x})$$

We wish to maximize the expected return while minimizing its variance. If we weight the expected return by w and variance by $1 - w$, we arrive at the following problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & -w\mathbf{x}^\top \boldsymbol{\mu} + (1-w)\mathbf{x}^\top \boldsymbol{\Sigma} \mathbf{x} \\ \text{subject to} \quad & \mathbf{x}^\top \mathbf{1} = b \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

This quadratic program (chapter 13) formulation is known as *Markowitz portfolio optimization*. The first constraint ensures that we allocate our full budget. The last constraint ensures we do not hold negative quantities of stock. The figure below shows how the optimal portfolio changes based on w.



Example 20.6. Considering both the expected value and the variance in portfolio optimization with

$$\boldsymbol{\mu} = [0.26, 0.08, 0.74]$$

and

$$\boldsymbol{\Sigma} = \begin{bmatrix} 0.21 & 0.03 & 0.01 \\ 0.03 & 0.06 & 0.04 \\ 0.01 & 0.04 & 0.94 \end{bmatrix}$$

Markowitz portfolio optimization is named for the American economist Harry Max Markowitz (1927–2023). H. Markowitz, "Portfolio Selection," *Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

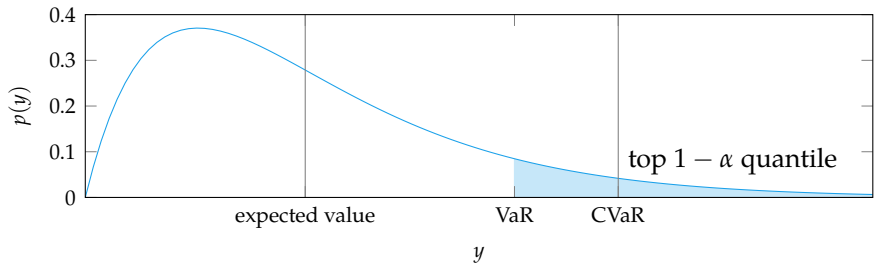


Figure 20.5. CVaR and VaR for a particular level α. CVaR is the expected value of the top $1-\alpha$ quantile, whereas the VaR is the lowest objective function value over the same quantile.

CVaR has some theoretical and computational advantages over VaR. CVaR is less sensitive to estimation errors in the distribution over the objective output. For example, if the cumulative distribution function is flat in some intervals, then VaR can jump with small changes in α. In addition, VaR does not account for costs beyond the α quantile, which is undesirable if there are rare outliers with very poor objective values.[16]

[16] For an overview of properties, see G. C. Pflug, "Some Remarks on the Value-at-Risk and the Conditional Value-at-Risk," in *Probabilistic Constrained Optimization: Methodology and Applications*, S. P. Uryasev, ed. Springer, 2000, pp. 272–281. and R. T. Rockafellar and S. Uryasev, "Conditional Value-at-Risk for General Loss Distributions," *Journal of Banking and Finance*, vol. 26, pp. 1443–1471, 2002.

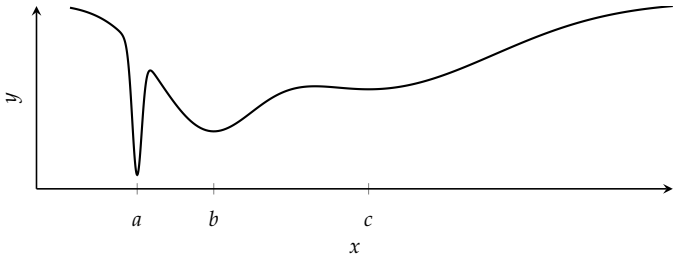
20.4 Summary

- Uncertainty in the optimization process can arise due to errors in the data, the models, or the optimization method itself.
- Accounting for these sources of uncertainty is important in ensuring robust designs.
- Optimization with respect to set-based uncertainty includes the minimax approach that assumes the worst-case and information-gap decision theory that finds a design robust to a maximally sized uncertainty set.
- Probabilistic approaches typically minimize the expected value, the variance, risk of infeasibility, value at risk, conditional value at risk, or a combination of these.

20.5 Exercises

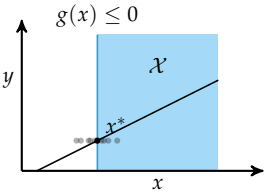
Exercise 20.1. Suppose we have zero-mean Gaussian noise in the input such that $f(x, z) = f(x + z)$. Consider the three points a, b, and c in the figure below:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

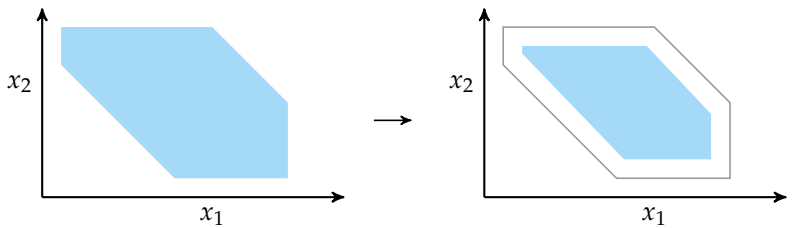


Which design point is best if we are minimizing the expected value minus the standard deviation?

Solution: The objective is to minimize $\mathbb{E}_z[f(x+z)] - \sqrt{\text{Var}_z[f(x+z)]}$. The first term, corresponding to the mean, is minimized at design point a. The second term, corresponding to the standard deviation, is also maximized at design point a because perturbations to the design at that location cause large variations in the output. The optimal design is thus $x^* = a$.



Exercise 20.2. Optima, such as the one depicted to the right, often lie on a constraint boundary and are thus sensitive to uncertainties that could cause them to become infeasible. One approach to overcome uncertainty with respect to feasibility is to make the constraints more stringent, reducing the size of the feasible region as shown below.



It is common to rewrite constraints of the form $g(\mathbf{x}) \leq g_{\text{max}}$ to $\gamma g(\mathbf{x}) \leq g_{\text{max}}$, where $\gamma > 1$ is a *factor of safety*. Optimizing such that the constraint values stay below g_{max}/γ provides an additional safety buffer.

Consider a beam with a square cross section thought to fail when the stresses exceed $\sigma_{\text{max}} = 1$. We wish to minimize the cross section $f(x) = x^2$, where x is the cross section length. The stress in the beam is also a function of the cross section length $g(x) = x^{-2}$. Plot the probability that the optimized design does not fail as the factor of safety varies from 1 to 2:

- Uncertainty in maximum stress, $g(x, z) = x^{-2} + z$
- Uncertainty in construction tolerance, $g(x, z) = (x + z)^{-2}$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

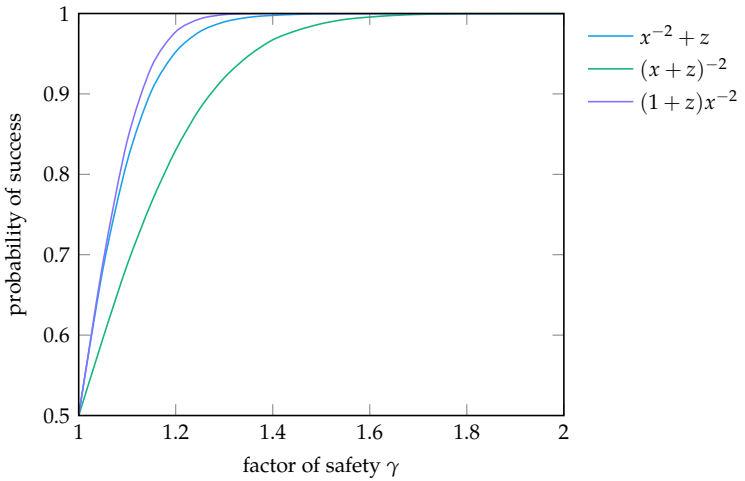
- Uncertainty in material properties, $g(x, z) = (1 + z)x^{-2}$

where z is zero-mean noise with variance 0.01.

Solution: The deterministic optimization problem is:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x^2 \\ \text{subject to} \quad & \gamma x^{-2} \leq 1 \end{aligned}$$

The optimal cross-section length as a function of the factor of safety is $x = \sqrt{\gamma}$. We can thus substitute $\sqrt{\gamma}$ for the cross-section length in each uncertainty formulation and evaluate the probability that the design does not fail. Note that all designs have a 50% chance of failure when the factor of safety is one, due to the symmetry of the normal distribution.



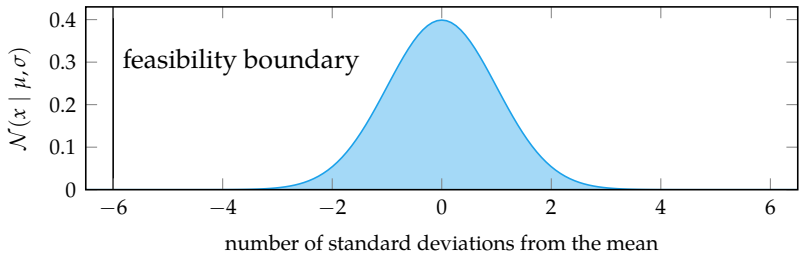
Exercise 20.3. The *six-sigma* method is a special case of statistical feasibility in which a production or industrial process is improved until its assumed Gaussian output violates design requirements only with outliers that exceed six standard deviations. This requirement is fairly demanding, as is illustrated below.

Consider the optimization problem

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & x_1 \\ \text{subject to} \quad & e^{x_1} \leq x_2 + z \leq 2e^{x_1} \end{aligned}$$

with $z \sim \mathcal{N}(0, 1)$. Find the optimal design x^* such that (x, z) is feasible for all $|z| \leq 6$.

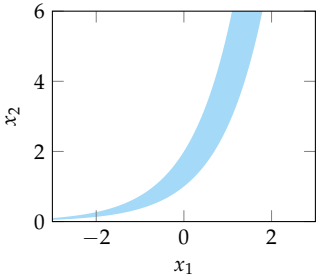
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



Solution: The figure on the right shows the noise-free feasible region. Without noise, the optimum lies with x_1 infinitely negative. We have noise and have chosen not to accept any outliers with magnitude greater than 6. Such outliers occur approximately $1.973 \times 10^{-7}\%$ of the time.

The feasible region for x_2 lies between e^{x_1} and $2e^{x_1}$. The noise is symmetric, so the most robust choice for x_2 is $1.5e^{x_1}$.

The width of the feasible region for x_2 is e^{x_1}, which increases with x_1. The objective function increases with x_1 as well, so the optimal x_1 is the lowest such that the width of the feasible region is at least 12. This results in $x_1 = \ln 12 \approx 2.485$ and $x_2 = 18$.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

21 Uncertainty Propagation

As discussed in the previous chapter, probabilistic approaches to optimization under uncertainty model some of the inputs to the objective function as a probability distribution. This chapter discusses how to propagate known input distributions to estimate quantities associated with the output distribution, such as the mean and variance of the objective function. There are a variety of approaches to *uncertainty propagation*, some based on mathematical concepts such as Monte Carlo, the Taylor series approximation, orthogonal polynomials, and Gaussian processes.[1] These approaches differ in the assumptions they make and the quality of their estimates.

[1] A variety of different uncertainty propagation methods are discussed by R. Ghanem, D. Higdon, and H. Owhadi, eds., *Handbook of Uncertainty Quantification*. Springer, 2017.

21.1 Sampling Methods

The mean and variance of the objective function at a particular design point can be approximated using *Monte Carlo integration*,[2] which approximates the integral using m samples $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}$, from the distribution p over $\mathcal{Z}$. These estimates are also called the *sample mean* and *sample variance*:

[2] Alternatively, quasi Monte Carlo integration can be used to produce estimates with faster convergence as discussed in chapter 16.

$$\mathbb{E}_{\mathbf{z}\sim p}[f(\mathbf{z})] \approx \hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{z}^{(i)}) \tag{21.1}$$

$$\text{Var}_{\mathbf{z}\sim p}[f(\mathbf{z})] \approx \hat{v} = \left(\frac{1}{m}\sum_{i=1}^{m} f(\mathbf{z}^{(i)})^2\right) - \hat{\mu}^2 \tag{21.2}$$

In the equation above, and for the rest of this chapter, we drop $\mathbf{x}$ from $f(\mathbf{x}, \mathbf{z})$ for notational convenience, but the dependency on $\mathbf{x}$ still exists. For each new design point $\mathbf{x}$ in our optimization process, we recompute the mean and variance.

A desirable property of this sampling-based approach is that p does not need to be known exactly. We can obtain samples directly from simulation or real-world experiments. A potential limitation of this approach is that many samples may be required before there is convergence to a suitable estimate. The variance of the sample mean for a normally distributed f is $\text{Var}[\hat{\mu}] = \nu/m$, where ν is the true variance of f. Thus, doubling the number of samples m tends to halve the variance of the sample mean.

21.2 Taylor Approximation

Another way to estimate $\hat{\mu}$ and $\hat{\nu}$ is to use the Taylor series approximation for f at a fixed design point $\mathbf{x}$.[3] For the moment, we will assume that the n components of $\mathbf{z}$ are uncorrelated and have finite variance. We will denote the mean of the distribution over $\mathbf{z}$ as $\boldsymbol{\mu}$ and the variances of the individual components of $\mathbf{z}$ as $\boldsymbol{\nu}$.[4] The following is the second-order Taylor series approximation of $f(\mathbf{z})$ at the point $\mathbf{z} = \boldsymbol{\mu}$:

$$\hat{f}(\mathbf{z}) = f(\boldsymbol{\mu}) + \sum_{i=1}^{n} \frac{\partial f}{\partial z_i}(z_i - \mu_i) + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial^2 f}{\partial z_i \partial z_j}(z_i - \mu_i)(z_j - \mu_j) \tag{21.3}$$

From this approximation, we can analytically compute estimates of the mean and variance of f:

$$\hat{\mu} = f(\boldsymbol{\mu}) + \frac{1}{2}\sum_{i=1}^{n} \frac{\partial^2 f}{\partial z_i^2}\nu_i \bigg|_{\mathbf{z}=\boldsymbol{\mu}} \tag{21.4}$$

$$\hat{\nu} = \sum_{i=1}^{n}\left(\frac{\partial f}{\partial z_i}\right)^2 \nu_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\frac{\partial^2 f}{\partial z_i \partial z_j}\right)^2 \nu_i \nu_j \bigg|_{\mathbf{z}=\boldsymbol{\mu}} \tag{21.5}$$

The higher-order terms can be neglected to obtain a first-order approximation:

$$\hat{\mu} = f(\boldsymbol{\mu}) \qquad \hat{\nu} = \sum_{i=1}^{n}\left(\frac{\partial f}{\partial z_i}\right)^2 \nu_i \bigg|_{\mathbf{z}=\boldsymbol{\mu}} \tag{21.6}$$

We can relax the assumption that the components of $\mathbf{z}$ are uncorrelated, but it makes the mathematics more complex. In practice, it can be easier to transform the random variables so that they are uncorrelated. We can transform a vector of n correlated random variables $\mathbf{c}$ with covariance matrix $\mathbf{C}$ into m uncorrelated

[3] For a derivation of the mean and variance of a general function of n random variables, see H. Benaroya and S. M. Han, *Probability Models in Engineering and Science*. Taylor & Francis, 2005.

[4] If the components of $\mathbf{z}$ are uncorrelated, then the covariance matrix is diagonal and $\boldsymbol{\nu}$ would be the vector composed of the diagonal elements.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

random variables $\mathbf{z}$ by multiplying by an orthogonal $m \times n$ matrix $\mathbf{T}$ containing the eigenvectors corresponding to the m largest eigenvalues of $\mathbf{C}$. We have $\mathbf{z} = \mathbf{T}\mathbf{c}$.[5]

The Taylor approximation method is implemented in algorithm 21.1. First- and second-order approximations are compared in example 21.1.

```
using ForwardDiff
function taylor_approx(f, μ, ν, secondorder=false)
    μhat = f(μ)
    ∇ = (z -> ForwardDiff.gradient(f, z))(μ)
    νhat = ∇.^2⋅ν
    if secondorder
        H = (z -> ForwardDiff.hessian(f, z))(μ)
        μhat += (diag(H)⋅ν)/2
        νhat += ν⋅(H.^2*ν)/2
    end
    return (μhat, νhat)
end
```

[5] It is also common to scale the outputs such that the covariance matrix becomes the identity matrix. This process is known as *whitening*. J. H. Friedman, "Exploratory Projection Pursuit," *Journal of the American Statistical Association*, vol. 82, no. 397, pp. 249–266, 1987.

Algorithm 21.1. A method for automatically computing the Taylor approximation of the mean and variance of objective function `f` at design point `x` with noise mean vector `μ` and variance vector `ν`. The Boolean parameter `secondorder` controls whether the first- or second-order approximation is computed.

21.3 Polynomial Chaos

Polynomial chaos is a method for fitting a polynomial to evaluations of $f(\mathbf{z})$ and using the resulting surrogate model to estimate the mean and variance. We will begin this section by discussing how polynomial chaos is used in the univariate case. We will then generalize the concept to multivariate functions and show how to obtain estimates of the mean and variance by integrating the function represented by our surrogate model.

21.3.1 Univariate

In one dimension, we approximate $f(z)$ with a surrogate model consisting of k polynomial basis functions, $b_1, \ldots, b_k$:

$$f(z) \approx \hat{f}(z) = \sum_{i=1}^{k} \theta_i b_i(z) \tag{21.7}$$

In contrast with the Monte Carlo methods discussed in section 21.1, our samples of z do not have to be randomly drawn from p. In fact, it may be desirable to obtain samples using one of the sampling plans discussed in chapter 16. We will discuss how to obtain the basis coefficients in section 21.3.2.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 21.1. The Taylor approximation applied to a univariate design problem with two-dimensional Gaussian noise.

Consider the objective function $f(x, z) = \sin(x + z_1)\cos(x + z_2)$, where z_1 and z_2 are zero-mean Gaussian noise with variances 0.1 and 0.2, respectively.

The first and second partial derivatives of f with respect to the zs are

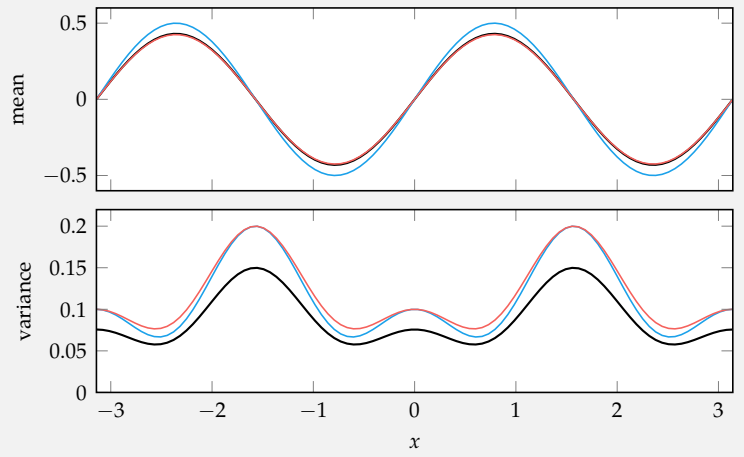
$$\frac{\partial f}{\partial z_1} = \cos(x + z_1)\cos(x + z_2) \qquad \frac{\partial^2 f}{\partial z_2^2} = -\sin(x + z_1)\cos(x + z_2)$$
$$\frac{\partial f}{\partial z_2} = -\sin(x + z_1)\sin(x + z_2) \qquad \frac{\partial^2 f}{\partial z_1 \partial z_2} = -\cos(x + z_1)\sin(x + z_2)$$
$$\frac{\partial^2 f}{\partial z_1^2} = -\sin(x + z_1)\cos(x + z_2)$$

which allow us to construct the Taylor approximation using equation (21.4) and equation (21.5):

$$\hat{\mu}(x) = 0.85\sin(x)\cos(x)$$
$$\hat{v}(x) = 0.1\cos^4(x) + 0.2\sin^4(x) + 0.045\sin^2(x)\cos^2(x)$$

We can use `taylor_approx` for a given `x` using:

```
taylor_approx(z->sin(x+z[1])*cos(x+z[2]), [0,0], [0.1,0.2])
```



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can use the surrogate model $\hat{f}$ to estimate the mean:

$$\begin{aligned}
\hat{\mu} &= \mathbb{E}\left[\hat{f}\right] && (21.8)\\
&= \int_{\mathcal{Z}} \hat{f}(z) p(z)\, dz && (21.9)\\
&= \int_{\mathcal{Z}} \sum_{i=1}^{k} \theta_i b_i(z) p(z)\, dz && (21.10)\\
&= \sum_{i=1}^{k} \theta_i \int_{\mathcal{Z}} b_i(z) p(z)\, dz && (21.11)\\
&= \theta_1 \int_{\mathcal{Z}} b_1(z) p(z)\, dz + \ldots + \theta_k \int_{\mathcal{Z}} b_k(z) p(z)\, dz && (21.12)
\end{aligned}$$

We can also estimate the variance:

$$\begin{aligned}
\hat{v} &= \mathbb{E}\left[\left(\hat{f} - \mathbb{E}\left[\hat{f}\right]\right)^2\right] && (21.13)\\
&= \mathbb{E}\left[\hat{f}^2 - 2\hat{f}\,\mathbb{E}\left[\hat{f}\right] + \mathbb{E}\left[\hat{f}\right]^2\right] && (21.14)\\
&= \mathbb{E}\left[\hat{f}^2\right] - \mathbb{E}\left[\hat{f}\right]^2 && (21.15)\\
&= \int_{\mathcal{Z}} \hat{f}(z)^2 p(z)\, dz - \mu^2 && (21.16)\\
&= \int_{\mathcal{Z}} \sum_{i=1}^{k} \sum_{j=1}^{k} \theta_i \theta_j b_i(z) b_j(z) p(z)\, dz - \mu^2 && (21.17)\\
&= \int_{\mathcal{Z}} \left(\sum_{i=1}^{k} \theta_i^2 b_i(z)^2 + 2 \sum_{i=2}^{k} \sum_{j=1}^{i-1} \theta_i \theta_j b_i(z) b_j(z) \right) p(z)\, dz - \mu^2 && (21.18)\\
&= \sum_{i=1}^{k} \theta_i^2 \int_{\mathcal{Z}} b_i(z)^2 p(z)\, dz + 2 \sum_{i=2}^{k} \sum_{j=1}^{i-1} \theta_i \theta_j \int_{\mathcal{Z}} b_i(z) b_j(z) p(z)\, dz - \mu^2 && (21.19)
\end{aligned}$$

The mean and variance can be efficiently computed if the basis functions are chosen to be *orthogonal* under p. Two basis functions b_i and b_j are orthogonal with respect to a probability density $p(z)$ if

$$\int_{\mathcal{Z}} b_i(z) b_j(z) p(z)\, dz = 0 \tag{21.20}$$

If the chosen basis functions are all orthogonal to one another and the first basis function is $b_1(z) = 1$, the mean is:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\hat{\mu} = \theta_1 \int_{\mathcal{Z}} b_1(z)p(z)\,dz + \theta_2 \int_{\mathcal{Z}} b_2(z)p(z)\,dz + \cdots + \theta_k \int_{\mathcal{Z}} b_k(z)p(z)\,dz \tag{21.21}$$

$$= \theta_1 \int_{\mathcal{Z}} b_1(z)^2 p(z)\,dz + \theta_2 \int_{\mathcal{Z}} b_1(z)b_2(z)p(z)\,dz + \cdots + \theta_k \int_{\mathcal{Z}} b_1(z)b_k(z)p(z)\,dz \tag{21.22}$$

$$= \theta_1 \int_{\mathcal{Z}} p(z)\,dz + 0 + \cdots + 0 \tag{21.23}$$

$$= \theta_1 \tag{21.24}$$

Similarly, the variance is:

$$\hat{v} = \sum_{i=1}^{k} \theta_i^2 \int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz + 2\sum_{i=2}^{k}\sum_{j=1}^{i-1} \theta_i\theta_j \int_{\mathcal{Z}} b_i(z)b_j(z)p(z)\,dz - \mu^2 \tag{21.25}$$

$$= \sum_{i=1}^{k} \theta_i^2 \int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz - \mu^2 \tag{21.26}$$

$$= \theta_1^2 \int_{\mathcal{Z}} b_1(z)^2 p(z)\,dz + \sum_{i=2}^{k} \theta_i^2 \int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz - \theta_1^2 \tag{21.27}$$

$$= \sum_{i=2}^{k} \theta_i^2 \int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz \tag{21.28}$$

The mean thus falls immediately from fitting a surrogate model to the observed data, and the variance can be very efficiently computed given the values $\int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz$ for a choice of basis functions and probability distribution.[6] Example 21.2 uses these procedures to estimate the mean and variance with different sample sizes.

[6] Integrals of this form can be efficiently computed using Gaussian quadrature, covered in appendix C.10.

Polynomial chaos approximates the function using orthogonal polynomial basis functions[7] with i in $1:k$ and $b_1 = 1$. All orthogonal polynomials satisfy the recurrence relation:

[7] The theory of orthogonal polynomials is surveyed by G. Szegö, *Orthogonal Polynomials*, 4th ed. American Mathematical Society, 2012.

$$b_{i+1}(z) = \begin{cases} (z - \alpha_i)b_i(z) & \text{for } i = 1 \\ (z - \alpha_i)b_i(z) - \beta_i b_{i-1}(z) & \text{for } i > 1 \end{cases} \tag{21.29}$$

with $b_1(z) = 1$ and weights

$$\alpha_i = \frac{\int_{\mathcal{Z}} z\, b_i(z)^2 p(z)\,dz}{\int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz} \tag{21.30}$$

$$\beta_i = \frac{\int_{\mathcal{Z}} b_i(z)^2 p(z)\,dz}{\int_{\mathcal{Z}} b_{i-1}(z)^2 p(z)\,dz} \tag{21.31}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

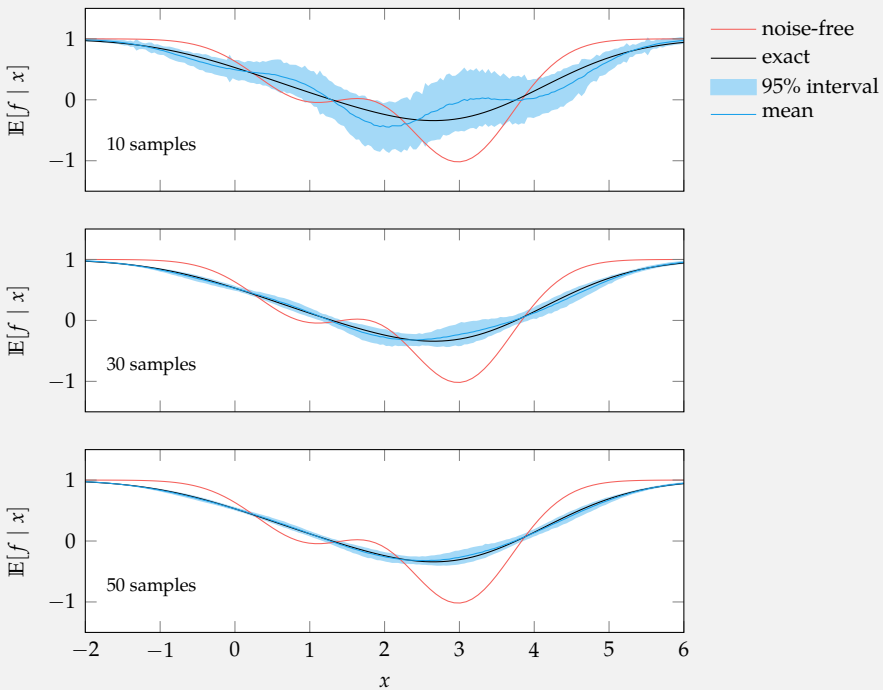
Example 21.2. Estimating the expected value of an unknown objective function using polynomial chaos. Hermite polynomials are defined in table 21.1.

Consider optimizing the (unknown) objective function

$$f(x,z) = 1 - e^{-(x+z-1)^2} - 2e^{-(x+z-3)^2}$$

with z known to be drawn from a zero-mean unit-Gaussian distribution.

The objective function, its true expected value, and estimated expected values with different sample counts are plotted below. The estimated expected value is computed using third-order Hermite polynomials.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The recurrence relation can be used to generate the basis functions. Each basis function b_i is a polynomial of degree $i - 1$. The basis functions for several common probability distributions are given in table 21.1, can be generated using the methods in algorithm 21.2, and are plotted in figure 21.1. Example 21.3 illustrates the effect the polynomial order has on the estimates of the mean and variance.

Distribution	Domain	Density	Name	Recursive Form $b_i(x) =$	Closed Form $b_i(x) =$
Uniform	$[-1, 1]$	$\frac{1}{2}$	Legendre	$\frac{2i-3}{i-1} x b_{i-1}(x) - \frac{i-2}{i-1} b_{i-2}(x)$	$\sum_{j=0}^{i-1} \binom{i-1}{j} \binom{i+j-1}{j} \left(\frac{x-1}{2}\right)^j$
Exponential	$[0, \infty)$	e^{-x}	Laguerre	$\frac{1}{i-1}((2i-3-x)b_{i-1} - (i-2)b_{i-2})$	$\sum_{j=0}^{i-1} \binom{i-1}{j} \frac{(-1)^j}{j!} x^j$
Unit Gaussian	$(-\infty, \infty)$	$\frac{1}{\sqrt{2\pi}} e^{-x^2/2}$	Hermite	$x b_{i-1}(x) - \frac{d}{dx} b_{i-1}(x)$	$(i-1)! \sum_{j=0}^{\lfloor (i-1)/2 \rfloor} \frac{(-1)^j}{j!(i-2j-1)!} \frac{x^{i-2j-1}}{2^j}$

Table 21.1. Orthogonal polynomial basis functions for several common probability distributions.

```
import Polynomials: Polynomial

function legendre(i)
    p = Polynomial([-1/2, 1/2])
    return sum(binomial(i-1,j)*binomial(i+j-1,j)*p^j for j in 0:i-1)
end
function laguerre(i)
    x = Polynomial([0,1])
    return sum(binomial(i-1,j)*(-x)^j/factorial(j) for j in 0:i-1)
end
function hermite(i)
    x = Polynomial([0,1])
    return factorial(i-1) * sum(
        (-1)^j / (factorial(j)*factorial(i-2*j-1)) *
            Polynomial([0,1])^(i-2j-1) / 2^j
        for j in 0:div(i-1,2))
end
```

Algorithm 21.2. Methods for constructing polynomial orthogonal basis functions, where `i` indicates the construction of b_i.

Basis functions for arbitrary probability density functions and domains can be constructed both analytically and numerically.[8] The *Stieltjes algorithm*[9] (algorithm 21.3) generates orthogonal polynomials using the recurrence relation in equation (21.29). Example 21.4 shows how the polynomial order affects the estimates of the mean and variance.

[8] The polynomials can be scaled by a nonzero factor. It is convention to set $b_1(x) = 1$.

[9] T. J. Stieltjes, "Quelques Recherches sur la Théorie des Quadratures Dites Mécaniques," *Annales Scientifiques de l'École Normale Supérieure*, vol. 1, pp. 409–426, 1884. in French. An overview in English is provided by W. Gautschi, *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, 2004.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

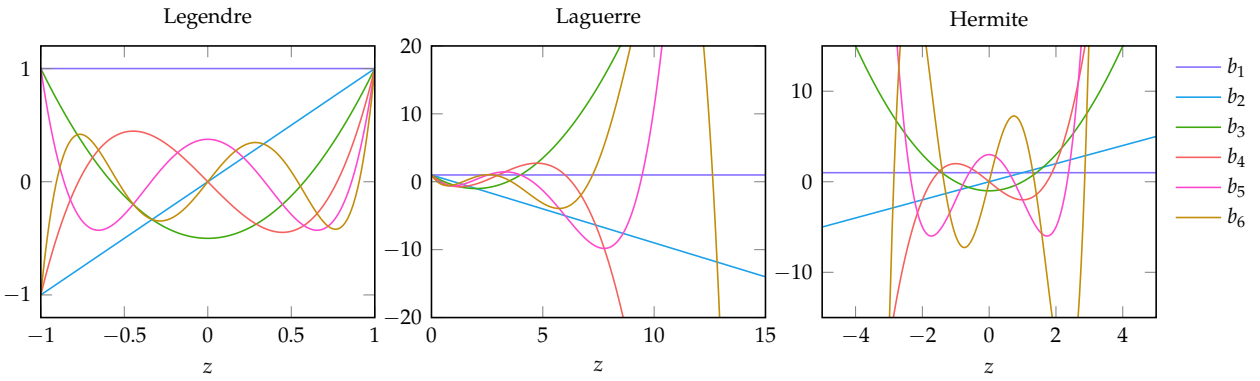


Figure 21.1. Orthogonal basis functions for uniform, exponential, and unit Gaussian distributions.

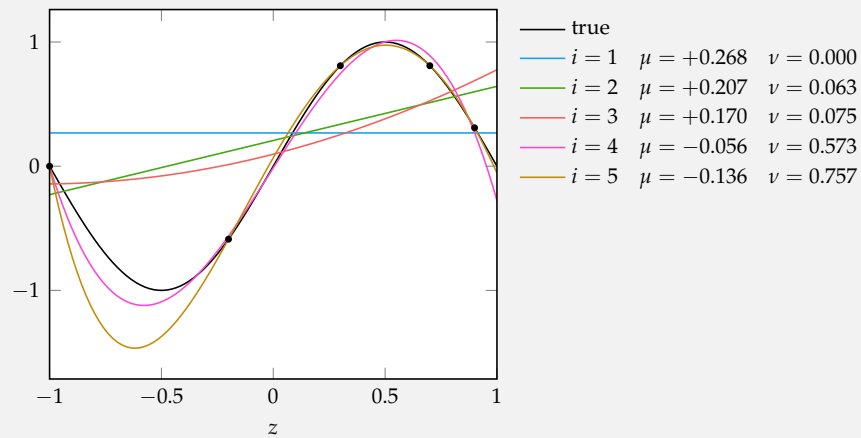
Example 21.3. Legendre polynomials used to estimate the mean and variance of a function with a uniformly distributed input.

Consider the function $f(z) = \sin(\pi z)$ with input z drawn from a uniform distribution over the domain $[-1, 1]$. The true mean and variance can be computed analytically:

$$\mu = \int_a^b f(z)p(z)\,dz = \int_{-1}^{1} \sin(\pi z)\frac{1}{2}\,dz = 0 \tag{21.32}$$

$$\nu = \int_a^b f(z)^2 p(z)\,dz - \mu^2 = \int_{-1}^{1} \sin^2(\pi z)\frac{1}{2}\,dz - 0 = \frac{1}{2} \tag{21.33}$$

Suppose we have five samples of f at $z = \{-1, -0.2, 0.3, 0.7, 0.9\}$. We can fit a Legendre polynomial to the data to obtain our surrogate model $\hat{f}$. Polynomials of different degrees yield:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function orthogonal_recurrence(bs, p, dom, ϵ=1e-6)
    i = length(bs)
    c1 = quadgk(z->z*bs[i](z)^2*p(z), dom..., atol=ϵ)[1]
    c2 = quadgk(z->  bs[i](z)^2*p(z), dom..., atol=ϵ)[1]
    α = c1 / c2
    if i > 1
        c3 = quadgk(z->bs[i-1](z)^2*p(z), dom..., atol=ϵ)[1]
        β = c2 / c3
        return Polynomial([-α, 1])*bs[i] - β*bs[i-1]
    else
        return Polynomial([-α, 1])*bs[i]
    end
end
```

Algorithm 21.3. The Stieltjes algorithm for constructing the next polynomial basis function b_{i+1} according to the orthogonal recurrence relation, where `bs` contains $\{b_1, \ldots, b_i\}$, `p` is the probability distribution, and `dom` is a tuple containing a lower and upper bound for z. The optional parameter `ϵ` controls the absolute tolerance of the numerical integration. We use the `Polynomials.jl` package.

21.3.2 *Coefficients*

The coefficients $\theta_1, \ldots, \theta_k$ in equation (21.7) can be inferred in two different ways. The first way is to fit the values of the samples from $\mathcal{Z}$ using the linear regression method discussed in section 17.3. The second way is to exploit the orthogonality of the basis functions, producing an integration term amenable to Gaussian quadrature.

We multiply each side of equation (21.7) by the jth basis and our probability density function and integrate:

$$f(z) \approx \sum_{i=1}^{k} \theta_i b_i(z) \tag{21.34}$$

$$\int_{\mathcal{Z}} f(z) b_j(z) p(z)\, dz \approx \int_{\mathcal{Z}} \left(\sum_{i=1}^{k} \theta_i b_i(z) \right) b_j(z) p(z)\, dz \tag{21.35}$$

$$= \sum_{i=1}^{k} \theta_i \int_{\mathcal{Z}} b_i(z) b_j(z) p(z)\, dz \tag{21.36}$$

$$= \theta_j \int_{\mathcal{Z}} b_j(z)^2 p(z)\, dz \tag{21.37}$$

where we made use of the orthogonality property from equation (21.20).

It follows that the jth coefficient is:

$$\theta_j = \frac{\int_{\mathcal{Z}} f(z) b_j(z) p(z)\, dz}{\int_{\mathcal{Z}} b_j(z)^2 p(z)\, dz} \tag{21.38}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 21.4. Orthogonal polynomials constructed using the Stieltjes method to estimate the mean and variance of a function with a random variable input.

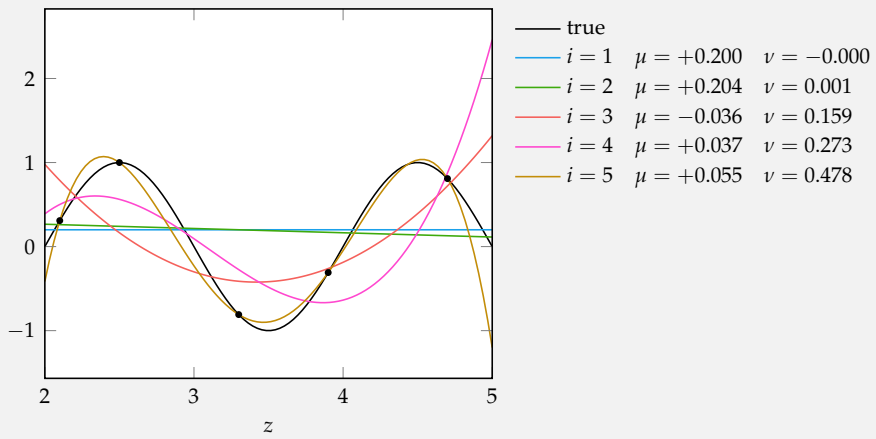
Consider the function $f(z) = \sin(\pi z)$ with input z drawn from a truncated Gaussian distribution with mean 3 and variance 1 over the domain $[2, 5]$. The true mean and variance are:

$$\mu = \int_a^b f(z)p(z)\,dz = \int_2^5 \sin(\pi z)p(z)\,dz \approx 0.104$$
$$\nu = \int_a^b f(z)^2 p(z)\,dz - \mu^2 = \int_2^5 \sin^2(\pi z)p(z)\,dz - 0.104^2 \approx 0.495$$

where the probability density of the truncated Gaussian is:

$$p(z) = \begin{cases} \frac{\mathcal{N}(z|3,1)}{\int_2^5 \mathcal{N}(\tau|3,1)\,d\tau} & \text{if } z \in [2,5] \\ 0 & \text{otherwise} \end{cases}$$

Suppose we have five samples of f at $z = \{2.1, 2.5, 3.3, 3.9, 4.7\}$. We can fit orthogonal polynomials to the data to obtain our surrogate model $\hat{f}$. Polynomials of different degrees yield:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The denominator of equation (21.38) typically has a known analytic solution or can be inexpensively precomputed. Calculating the coefficient thus primarily requires solving the integral in the numerator, which can be done numerically using Gaussian quadrature.[10]

[10] Gaussian quadrature is implemented in `QuadGK.jl` via the `quadgk` function, and is covered in appendix C.10. Quadrature rules can also be obtained using the eigenvalues and eigenvectors of a tri-diagonal matrix formed using the coefficients α_i and β_i from equations (21.30) and (21.31). G. H. Golub and J. H. Welsch, "Calculation of Gauss Quadrature Rules," *Mathematics of Computation*, vol. 23, no. 106, pp. 221–230, 1969.

21.3.3 *Multivariate*

Polynomial chaos can be applied to functions with multiple random inputs. Multivariate basis functions over m variables are constructed as a product over univariate orthogonal polynomials:

$$b_i(\mathbf{z}) = \prod_{j=1}^{m} b_{a_j}(z_j) \tag{21.39}$$

where $\mathbf{a}$ is an assignment vector that assigns the a_jth basis function to the jth random component. This basis function construction is demonstrated in example 21.5.

Consider a three-dimensional polynomial chaos model of which one of the multidimensional basis functions is $b(\mathbf{z}) = b_3(z_1)b_1(z_2)b_3(z_3)$. The corresponding assignment vector is $\mathbf{a} = [3, 1, 3]$.

Example 21.5. Constructing a multivariate polynomial chaos basis function using equation (21.39).

A common method for constructing multivariate basis functions is to generate univariate orthogonal polynomials for each random variable and then to construct a multivariate basis function for every possible combination.[11] This procedure is implemented in algorithm 21.4. Constructing basis functions in this manner assumes that the variables are independent. Interdependence can be resolved using the same transformation discussed in section 21.2.

[11] Here the number of multivariate basis functions grows exponentially in the number of variables.

```
function polynomial_chaos_bases(bases1d)
    bases = []
    for a in Iterators.product(bases1d...)
        push!(bases,
            z -> prod(b(z[i]) for (i,b) in enumerate(a)))
    end
    return bases
end
```

Algorithm 21.4. A method for constructing multivariate basis functions where `bases1d` contains lists of univariate orthogonal basis functions for each random variable.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A multivariate polynomial chaos approximation with k basis functions is still a linear combination of terms

$$f(\mathbf{z}) \approx \hat{f}(\mathbf{z}) = \sum_{i=1}^{k} \theta_i b_i(\mathbf{z}) \tag{21.40}$$

where the mean and variance can be computed using the equations in section 21.3.1, provided that $b_1(\mathbf{z}) = 1$.

21.4 Bayesian Monte Carlo

Gaussian processes, covered in chapter 18, are probability distributions over functions. They can be used as surrogates for stochastic objective functions. We can incorporate prior information, such as the expected smoothness of the objective function, in a process known as *Bayesian Monte Carlo* or *Bayes-Hermite Quadrature*.

Consider a Gaussian process fit to several points with the same value for the design point $\mathbf{x}$ but different values for the uncertain point $\mathbf{z}$. The Gaussian process obtained is a distribution over functions based on the observed data. When obtaining the expected value through integration, we must consider the expected value of the functions in the probability distribution represented by the Gaussian process $p(\hat{f})$:

$$\begin{aligned}
\mathbb{E}_{\mathbf{z}\sim p}[f] &\approx \mathbb{E}_{\hat{f}\sim p(\hat{f})}\left[\mathbb{E}_{\mathbf{z}\sim p}\left[\hat{f}\right]\right] && (21.41)\\
&= \int_{\hat{\mathcal{F}}} \left(\int_{\mathcal{Z}} \hat{f}(\mathbf{z}) p(\mathbf{z})\, d\mathbf{z} \right) p(\hat{f})\, d\hat{f} && (21.42)\\
&= \int_{\mathcal{Z}} \left(\int_{\hat{\mathcal{F}}} \hat{f}(\mathbf{z}) p(\hat{f})\, d\hat{f} \right) p(\mathbf{z})\, d\mathbf{z} && (21.43)\\
&= \int_{\mathcal{Z}} \hat{\mu}(\mathbf{z}) p(\mathbf{z})\, d\mathbf{z} && (21.44)
\end{aligned}$$

where $\hat{\mu}(\mathbf{z})$ is the predicted mean under the Gaussian process and $\hat{\mathcal{F}}$ is the space of functions. The variance of the estimate is

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\text{Var}_{\mathbf{z}\sim p}[f] \approx \text{Var}_{\hat{f}\sim p(\hat{f})}\left[\mathbb{E}_{\mathbf{z}\sim p}\left[\hat{f}\right]\right] \tag{21.45}$$

$$= \int_{\hat{\mathcal{F}}}\left(\int_{\mathcal{Z}} \hat{f}(\mathbf{z})p(\mathbf{z})\,d\mathbf{z} - \int_{\mathcal{Z}} \mathbb{E}_{\hat{f}\sim p(\hat{f})}[\hat{f}(\mathbf{z}')]p(\mathbf{z}')\,d\mathbf{z}'\right)^2 p(\hat{f})\,d\hat{f} \tag{21.46}$$

$$= \int_{\mathcal{Z}}\int_{\mathcal{Z}}\int_{\hat{\mathcal{F}}}\left[\hat{f}(\mathbf{z}) - \mathbb{E}_{\hat{f}\sim p(\hat{f})}[\hat{f}(\mathbf{z})]\right]\left[\hat{f}(\mathbf{z}') - \mathbb{E}_{\hat{f}\sim p(\hat{f})}[\hat{f}(\mathbf{z}')]\right]p(\hat{f})\,d\hat{f}p(\mathbf{z})p(\mathbf{z}')\,d\mathbf{z}\,d\mathbf{z}' \tag{21.47}$$

$$= \int_{\mathcal{Z}}\int_{\mathcal{Z}} \text{Cov}(\hat{f}(\mathbf{z}), \hat{f}(\mathbf{z}'))p(\mathbf{z})p(\mathbf{z}')\,d\mathbf{z}\,d\mathbf{z}' \tag{21.48}$$

where Cov is the posterior covariance under the Gaussian process:

$$\text{Cov}(\hat{f}(\mathbf{z}), \hat{f}(\mathbf{z}')) = k(\mathbf{z},\mathbf{z}') - k(\mathbf{z},Z)\mathbf{K}(Z,Z)^{-1}k(Z,\mathbf{z}') \tag{21.49}$$

where Z contains the observed inputs.

Analytic expressions exist for the mean and variance for the special case where $\mathbf{z}$ is Gaussian.[12] Under a Gaussian kernel,

$$k(\mathbf{x},\mathbf{x}') = \exp\left(-\frac{1}{2}\sum_{i=1}^{n}\frac{(x_i - x_i')^2}{w_i^2}\right) \tag{21.50}$$

the mean for Gaussian uncertainty $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{z}, \boldsymbol{\Sigma}_\mathbf{z})$ is

$$\mathbb{E}_{\mathbf{z}\sim p}[f] \approx \mathbb{E}_{\hat{f}\sim p(\hat{f})}\left[\mathbb{E}_{\mathbf{z}\sim p}\left[\hat{f}\right]\right] = \mathbf{q}^\top\mathbf{K}^{-1}\mathbf{y} \tag{21.51}$$

with

$$q_i = |\mathbf{W}^{-1}\boldsymbol{\Sigma}_\mathbf{z} + \mathbf{I}|^{-1/2}\exp\left(-\frac{1}{2}\left(\boldsymbol{\mu}_\mathbf{z} - \mathbf{z}^{(i)}\right)^\top(\boldsymbol{\Sigma}_\mathbf{z} + \mathbf{W})^{-1}\left(\boldsymbol{\mu}_\mathbf{z} - \mathbf{z}^{(i)}\right)\right) \tag{21.52}$$

where $\mathbf{W} = \text{diag}[w_1^2, \ldots, w_n^2]$, and we have constructed our Gaussian process using samples $(\mathbf{z}^{(i)}, y_i)$ for i in $1:m$.[13]

The variance is

$$\text{Var}_{\mathbf{z}\sim p}[f] = |2\mathbf{W}^{-1}\boldsymbol{\Sigma}_\mathbf{z} + \mathbf{I}|^{-1/2} - \mathbf{q}^\top\mathbf{K}^{-1}\mathbf{q} \tag{21.53}$$

Even when the analytic expressions are not available, there are many problems for which numerically evaluating the expectation is sufficiently inexpensive that the Gaussian process approach is better than a Monte Carlo estimation.

Bayesian Monte Carlo is implemented in algorithm 21.5 and is worked out in example 21.6.

[12] It is also required that the covariance function obey the *product correlation rule*; that is, it can be written as the product of a univariate positive-definite function r:

$$k(\mathbf{x},\mathbf{x}') = \prod_{i=1}^{n} r(x_i - x_i')$$

Analytic results exist for polynomial kernels and mixtures of Gaussians. C. E. Rasmussen and Z. Ghahramani, "Bayesian Monte Carlo," in *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[13] See A. Girard, C. E. Rasmussen, J. Q. Candela, and R. Murray-Smith, "Gaussian Process Priors with Uncertain Inputs—Application to Multiple-Step Ahead Time Series Forecasting," in *Advances in Neural Information Processing Systems (NIPS)*, 2003.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function bayesian_monte_carlo(GP, w, μz, Σz)
    W = Matrix(Diagonal(w.^2))
    invK = inv(K(GP.X, GP.X, GP.k))
    q = [exp(-((z-μz)⋅(inv(W+Σz)*(z-μz)))/2) for z in GP.X]
    q .*= (det(W\Σz + I))^(-0.5)
    μ = q'*invK*GP.y
    ν = (det(2(W\Σz) + I))^(-0.5) - (q'*invK*q)[1]
    return (μ, ν)
end
```

Algorithm 21.5. A method for obtaining the Bayesian Monte Carlo estimate for the expected value of a function under a Gaussian process `GP` with a Gaussian kernel with weights `w`, where the variables are drawn from a normal distribution with mean `μz` and covariance `Σz`.

21.5 Summary

- The expected value and variance of the objective function are useful when optimizing problems involving uncertainty, but computing these quantities reliably can be challenging.
- One of the simplest approaches is to estimate the moments using sampling in a process known as Monte Carlo integration.
- Other approaches, such as the Taylor approximation, use knowledge of the objective function's partial derivatives.
- Polynomial chaos is a powerful uncertainty propagation technique based on orthogonal polynomials.
- Bayesian Monte Carlo uses Gaussian processes to efficiently arrive at the moments with analytic results for Gaussian kernels.

21.6 Exercises

Exercise 21.1. Suppose we draw a sample from a univariate Gaussian distribution. What is the probability that our sample falls within one standard deviation of the mean? What is the probability that our sample is less than one standard deviation above the mean?

Solution: The cumulative distribution function can be used to calculate these values:

```
julia> using Distributions
julia> N = Normal(0,1);
julia> cdf(N, 1) - cdf(N, -1)
0.6826894921370861
julia> cdf(N, 1)
0.841344746068543
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 21.6. An example of using Bayesian Monte Carlo to estimate the expected value and variance of a function.

The figure compares the Bayesian Monte Carlo method to the sample mean for estimating the expected value of a function. The same randomly sampled $\mathbf{z}$ values generated for each evaluated x were input to each method.

Consider again estimating the expected value and variance of $f(x, z) = \sin(x + z_1)\cos(x + z_2)$, where z_1 and z_2 are zero-mean Gaussian noise with variances 1 and $1/2$, respectively: $\boldsymbol{\mu}_\mathbf{z} = [0, 0]$ and $\boldsymbol{\Sigma}_\mathbf{z} = \text{diag}([1, 1/2])$.

We use Bayesian Monte Carlo with a Gaussian kernel with unit weights $\mathbf{w} = [1, 1]$ for $x = 0$ with samples $Z = \{[0, 0], [1, 0], [-1, 0], [0, 1], [0, -1]\}$.

We compute:

$$\mathbf{W} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

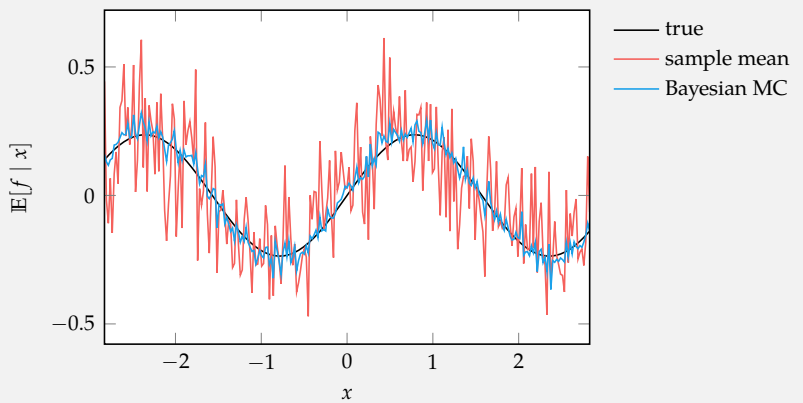
$$\mathbf{K} = \begin{bmatrix} 1 & 0.607 & 0.607 & 0.607 & 0.607 \\ 0.607 & 1 & 0.135 & 0.368 & 0.368 \\ 0.607 & 0.135 & 1 & 0.368 & 0.368 \\ 0.607 & 0.368 & 0.368 & 1 & 0.135 \\ 0.607 & 0.368 & 0.368 & 0.135 & 1 \end{bmatrix}$$

$$\mathbf{q} = [0.577, 0.450, 0.450, 0.414, 0.414]$$

$$\mathbb{E}_{\mathbf{z}\sim p}[f] = 0.0$$

$$\text{Var}_{\mathbf{z}\sim p}[f] = 0.005$$

Below we plot the expected value as a function of x using the same approach with ten random samples of $\mathbf{z}$ at each point.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Thus, our sample falls within one standard deviation of the mean about 68.3% of the time and is less than one standard deviation above the mean 84.1% of the time.

Exercise 21.2. Let $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ be a random sample of independent, identically distributed values of size m from a distribution with mean μ and variance ν. Show that the variance of the sample mean $\text{Var}(\hat{\mu})$ is ν/m.

Solution: We begin by substituting in the definition of the sample mean:

$$\begin{aligned}\text{Var}(\hat{\mu}) &= \text{Var}\left(\frac{x^{(1)} + x^{(2)} + \cdots + x^{(m)}}{m}\right) \\ &= \text{Var}\left(\frac{1}{m}x^{(1)} + \frac{1}{m}x^{(2)} + \cdots + \frac{1}{m}x^{(m)}\right)\end{aligned}$$

The variance of the sum of two independent variables is the sum of the variances of the two variables. It follows that:

$$\begin{aligned}\text{Var}(\hat{\mu}) &= \text{Var}\left(\frac{1}{m}x^{(1)}\right) + \text{Var}\left(\frac{1}{m}x^{(2)}\right) + \cdots + \text{Var}\left(\frac{1}{m}x^{(m)}\right) \\ &= \frac{1}{m^2}\text{Var}\left(x^{(1)}\right) + \frac{1}{m^2}\text{Var}\left(x^{(2)}\right) + \cdots + \frac{1}{m^2}\text{Var}\left(x^{(m)}\right) \\ &= \frac{1}{m^2}(\nu + \nu + \cdots + \nu) \\ &= \frac{1}{m^2}(m\nu) \\ &= \frac{\nu}{m}\end{aligned}$$

Exercise 21.3. Derive the recurrence relation equation (21.29) that is satisfied by all orthogonal polynomials.

Solution: The three-term recurrence relation for orthogonal polynomials is central to their construction and use. A key to the derivation is noticing that a multiple of z can be shifted from one basis to the other:

$$\int_{\mathcal{Z}} (zb_i(z))b_j(z)p(z)\,dz = \int_{\mathcal{Z}} b_i(z)\Big(zb_j(z)\Big)p(z)\,dz$$

We must show that

$$b_{i+1}(z) = \begin{cases}(z - \alpha_i)b_i(z) & \text{for } i = 1 \\ (z - \alpha_i)b_i(z) - \beta_i b_{i-1}(z) & \text{for } i > 1\end{cases}$$

produces orthogonal polynomials.

We notice that $b_{i+1} - zb_i$ is a polynomial of degree at most i, allowing us to write it as a linear combination of the first i orthogonal polynomials:

$$b_{i+1}(z) - zb_i(z) = -\alpha_i b_i(z) - \beta_i b_{i-1}(z) + \sum_{j=0}^{i-2} \gamma_{ij} b_j(z)$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

for constants α_i, β_i, and γ_{ij}.

Multiplying both sides by b_i and p and then integrating yields:

$$\int_{\mathcal{Z}} (b_{i+1}(z) - z b_i(z)) b_i(z) p(z)\, dz = \int_{\mathcal{Z}} \left(-\alpha_i b_i(z) - \beta_i b_{i-1}(z) + \sum_{j=0}^{i-2} \gamma_{ij} b_j(z) \right) b_i(z) p(z)\, dz$$

$$\begin{aligned}\int_{\mathcal{Z}} b_{i+1}(z) b_i(z) p(z)\, dz - \int_{\mathcal{Z}} z b_i(z) b_i(z) p(z)\, dz = &- \int_{\mathcal{Z}} \alpha_i b_i(z) b_i(z) p(z)\, dz - \\ &- \int_{\mathcal{Z}} \beta_i b_{i-1}(z) b_i(z) p(z)\, dz + \\ &+ \sum_{j=0}^{i-2} \int_{\mathcal{Z}} \gamma_{ij} b_j(z) b_i(z) p(z)\, dz \\ - \int_{\mathcal{Z}} z b_i^2(z) p(z)\, dz = &- \alpha_i \int_{\mathcal{Z}} b_i^2(z) p(z)\, dz\end{aligned}$$

producing our expression for α_i:

$$\alpha_i = \frac{\int_{\mathcal{Z}} z b_i^2(z) p(z)\, dz}{\int_{\mathcal{Z}} b_i^2(z) p(z)\, dz}$$

The expression for β_i with $i \geq 1$ is obtained instead by multiplying both sides by b_{i-1} and p and then integrating.

Multiplying both sides by b_k, with $k < i - 1$, similarly produces:

$$- \int_{\mathcal{Z}} z b_i(z) b_k(z) p(z)\, dz = \gamma_{ik} \int_{\mathcal{Z}} b_k^2(z) p(z)\, dz$$

The shift property can be applied to yield:

$$\int_{\mathcal{Z}} z b_i(z) b_k(z) p(z)\, dz = \int_{\mathcal{Z}} b_i(z) (z b_k(z)) p(z)\, dz = 0$$

as $z b_k(z)$ is a polynomial of at most order $i - 1$, and, by orthogonality, the integral is zero. It follows that all γ_{ik} are zero, and the three term recurrence relation is established.

Exercise 21.4. Suppose we have fitted a polynomial chaos model of an objective function $f(\mathbf{x}, \mathbf{z})$ for a particular design point $\mathbf{x}$ using m evaluations with $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}$. Derive an expression for estimating the partial derivative of the polynomial chaos coefficients with respect to a design component x_i.

Solution: We can derive a gradient approximation using the partial derivative of f with respect to a design component x_i:

$$\frac{\partial}{\partial x_i} f(\mathbf{x}, \mathbf{z}) \approx b_1(\mathbf{z}) \frac{\partial}{\partial x_i} \theta_1(\mathbf{x}) + \cdots + b_k(\mathbf{z}) \frac{\partial}{\partial x_i} \theta_k(\mathbf{x})$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If we have m samples, we can write these partial derivatives in matrix form:

$$\begin{bmatrix} \frac{\partial}{\partial x_i} f(\mathbf{x}, \mathbf{z}^{(1)}) \\ \vdots \\ \frac{\partial}{\partial x_i} f(\mathbf{x}, \mathbf{z}^{(m)}) \end{bmatrix} \approx \begin{bmatrix} b_1(\mathbf{z}^{(1)}) & \cdots & b_k(\mathbf{z}^{(1)}) \\ \vdots & \ddots & \vdots \\ b_1(\mathbf{z}^{(m)}) & \cdots & b_k(\mathbf{z}^{(m)}) \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x_i} \theta_1(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_i} \theta_k(\mathbf{x}) \end{bmatrix}$$

We can solve for approximations of $\frac{\partial}{\partial x_i}\theta_1(\mathbf{x}), \ldots, \frac{\partial}{\partial x_i}\theta_k(\mathbf{x})$ using the pseudoinverse:

$$\begin{bmatrix} \frac{\partial}{\partial x_i} \theta_1(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_i} \theta_k(\mathbf{x}) \end{bmatrix} \approx \begin{bmatrix} b_1(\mathbf{z}^{(1)}) & \cdots & b_k(\mathbf{z}^{(1)}) \\ \vdots & \ddots & \vdots \\ b_1(\mathbf{z}^{(m)}) & \cdots & b_k(\mathbf{z}^{(m)}) \end{bmatrix}^+ \begin{bmatrix} \frac{\partial}{\partial x_i} f(\mathbf{x}, \mathbf{z}^{(1)}) \\ \vdots \\ \frac{\partial}{\partial x_i} f(\mathbf{x}, \mathbf{z}^{(m)}) \end{bmatrix}$$

Exercise 21.5. Consider an objective function $f(\mathbf{x}, \mathbf{z})$ with design variables $\mathbf{x}$ and random variables $\mathbf{z}$. As discussed in chapter 20, optimization under uncertainty often involves minimizing a linear combination of the estimated mean and variance:

$$f_{\text{mod}}(\mathbf{x}, \mathbf{z}) = \alpha \hat{\mu}(\mathbf{x}) + (1 - \alpha)\hat{v}(\mathbf{x})$$

How can one use polynomial chaos to estimate the gradient of f_{mod} with respect to a design variable $\mathbf{x}$?

Solution: The estimated mean and variance have coefficients that depend on the design variables:

$$\begin{aligned} \hat{\mu}(\mathbf{x}) &= \theta_1(\mathbf{x}) \\ \hat{v}(\mathbf{x}) &= \sum_{i=2}^{k} \theta_i^2(\mathbf{x}) \int_{\mathcal{Z}} b_i(\mathbf{z})^2 p(\mathbf{z})\, d\mathbf{z} \end{aligned}$$

The partial derivative of f_{mod} with respect to the ith design component is

$$\frac{\partial}{\partial x_i} f_{\text{mod}}(\mathbf{x}) = \alpha \frac{\partial \theta_1(\mathbf{x})}{\partial x_i} + 2(1 - \alpha) \sum_{j=2}^{k} \theta_j(\mathbf{x}) \frac{\partial \theta_j(\mathbf{x})}{\partial x_i} \int_{\mathcal{Z}} b_j(\mathbf{z})^2 p(\mathbf{z})\, d\mathbf{z}$$

This computation requires the gradient of the coefficients with respect to $\mathbf{x}$, which is estimated in exercise 21.4.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

22 *Discrete Optimization*

Previous chapters have focused on optimizing problems involving design variables that are continuous. Many problems, however, have design variables that are naturally discrete, such as manufacturing problems involving mechanical components that come in fixed sizes or navigation problems involving choices between discrete paths. A *discrete optimization* problem has constraints such that the design variables must be chosen from a discrete set. Some discrete optimization problems have infinite design spaces, and others are finite.[1] Even for finite problems, where we could in theory enumerate every possible solution, it is generally not computationally feasible to do so in practice. This chapter discusses both exact and approximate approaches to solving discrete optimization problems that avoid enumeration. Many of the methods covered earlier, such as simulated annealing and genetic programming, can easily be adapted for discrete optimization problems, but we will focus this chapter on categories of techniques we have not yet discussed.

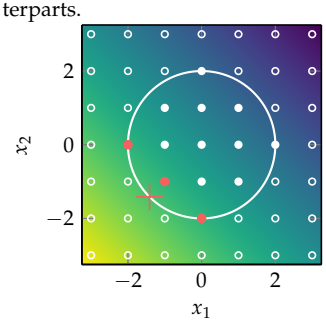
[1] Discrete optimization with a finite design space is sometimes referred to as *combinatorial optimization*. For a review, see B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 6th ed. Springer, 2018.

Example 22.1. Discrete versions of problems constrain the solution, often resulting in worse solutions than their continuous counterparts.

Discrete optimization constrains the design to be integral. Consider the problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & x_1 + x_2 \\ \text{subject to} \quad & \|\mathbf{x}\| \leq 2 \\ & \mathbf{x} \text{ is integral} \end{aligned}$$

The optimum in the continuous case is $\mathbf{x}^* = [-\sqrt{2}, -\sqrt{2}]$ with a value of $y = -2\sqrt{2} \approx -2.828$. If x_1 and x_2 are constrained to be integer-valued, then the best we can do is to have $y = -2$ with $\mathbf{x}^* \in \{[-2, 0], [-1, -1], [0, -2]\}$.



22.1 Integer Programs

An *integer program* is a linear program[2] with integral constraints. By integral constraints, we mean that the design variables must come from the set of integers.[3] Integer programs are sometimes referred to as *integer linear programs* to emphasize the assumption that the objective function and constraints are linear.

An integer program in standard form is expressed as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x} \in \mathbb{Z}^n \end{aligned} \tag{22.1}$$

where $\mathbb{Z}^n$ is the set of n-dimensional integral vectors. As with linear programs, $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, and $\mathbf{b} \in \mathbb{R}^m$.

Like linear programs, integer programs are often solved in equality form. Transforming an integer program to equality form often requires adding additional slack variables $\mathbf{s}$ that do not need to be integral. Thus, the equality form for integral programs is:

$$\begin{aligned} \underset{\mathbf{x},\mathbf{s}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{s} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{s} \geq \mathbf{0} \\ & \mathbf{x} \in \mathbb{Z}^n \end{aligned} \tag{22.2}$$

More generally, a *mixed-integer program* (algorithm 22.1) includes both continuous and discrete design components. Such a problem, in equality form, is expressed as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x}_{\mathcal{D}} \in \mathbb{Z}^{|\mathcal{D}|} \end{aligned} \tag{22.3}$$

where $\mathcal{D}$ is a set of indices corresponding to the design variables that are constrained to be discrete. Here, $\mathbf{x} = [\mathbf{x}_{\mathcal{D}}, \mathbf{x}_{\mathcal{C}}]$, where $\mathbf{x}_{\mathcal{D}}$ represents the vector of discrete design variables and $\mathbf{x}_{\mathcal{C}}$ the vector of continuous design variables.

[2] See chapter 12.

[3] Integer programming is a very mature field, with applications in operations research, communications networks, task scheduling, and other disciplines. Modern solvers, such as Gurobi and CPLEX, can routinely handle problems with millions of variables. There are packages for Julia that provide access to Gurobi, CPLEX, and a variety of other solvers.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
mutable struct MixedIntegerProgram
    A # minimize c⋅x
    b # subject to Ax = b
    c #             x ≥ 0
    D #             x[D] ∈ ℤᴰ
end
```

Algorithm 22.1. A mixed-integer linear program type that reflects equation (22.3). Here, `D` is the set of discrete design indices.

22.2 *Rounding*

A common approach to discrete optimization is to *relax* the constraint that the design points must come from a discrete set. The advantage of this relaxation is that we can use techniques, like gradient descent or linear programming, that take advantage of the continuous nature of the objective function to direct the search. After a continuous solution is found, the design variables are *rounded* to the nearest discrete design. An alternative is *randomized rounding*, where the design variables are rounded probabilistically up or down to a discrete design.[4]

There are potential issues with rounding. Rounding might result in an infeasible design point, as shown in figure 22.1. Even if rounding results in a feasible point, it may be far from optimal, as shown in figure 22.2. The addition of the discrete constraint will typically worsen the objective value as illustrated in example 22.1. However, for some problems, we can show the relaxed solution is close to the optimal discrete solution.

We can solve integer programs using rounding by removing the integer constraint, solving the corresponding linear program, or LP, and then rounding the solution to the nearest integer. This method is implemented in algorithm 22.2.

[4] P. Raghavan and C. D. Tompson, "Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.

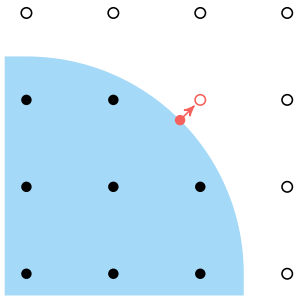


Figure 22.1. Rounding can produce an infeasible design point.

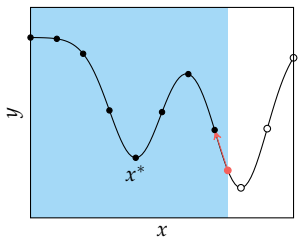


Figure 22.2. The nearest feasible discrete design may be significantly worse than the best feasible discrete design.

```
relax(MIP) = LinearProgram(MIP.A, MIP.b, MIP.c)
round_ip(MIP) = round.(Int, minimize_lp(relax(MIP)).x)
```

Algorithm 22.2. Relaxing a mixed-integer linear program into a linear program and then rounding.

We can show that rounding the continuous solution for a constraint $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ when $\mathbf{A}$ is integral is never too far from the optimal integral solution.[5] A bound can be derived based on the determinants of the submatrices of $\mathbf{A}$, where a *submatrix* is a matrix obtained by deleting rows and/or columns of another matrix. If $\mathbf{x}_c^*$ is an optimal solution of the LP with $m \times n$ matrix $\mathbf{A}$, then there exists an optimal discrete solution $\mathbf{x}_d^*$ with $\left\|\mathbf{x}_c^* - \mathbf{x}_d^*\right\|_\infty$ less than or equal to n times the maximum absolute value of the determinants of the submatrices of $\mathbf{A}$.

[5] W. Cook, A. M. Gerards, A. Schrijver, and É. Tardos, "Sensitivity Theorems in Integer Linear Programming," *Mathematical Programming*, vol. 34, no. 3, pp. 251–264, 1986.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The vector $\mathbf{c}$ need not be integral for an LP to have an optimal integral solution because the feasible region is purely determined by $\mathbf{A}$ and $\mathbf{b}$. Some approaches use the dual formulation for the LP, which has a feasible region dependent on $\mathbf{c}$, in which case having an integral $\mathbf{c}$ is also required.

In the special case of *totally unimodular* integer programs, where $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}$ have all integer entries and $\mathbf{A}$ is totally unimodular, the simplex algorithm is guaranteed to return an integer solution. A matrix is totally unimodular if the determinant of every square submatrix is 0, 1, or -1. If a totally unimodular matrix is invertible, then its inverse is also integral. As a result, every vertex solution of a totally unimodular integer program is integral.

Several matrices and their total unimodularity are discussed in example 22.2. Methods for determining whether a matrix or an integer linear program is totally unimodular are given in algorithm 22.3.

Example 22.2. Examples of matrices that are and are not totally unimodular.

Consider the following matrices:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & 0 \\ 0 & 1 & 1 & 0 & -1 \end{bmatrix}$$

The left matrix is not totally unimodular because

$$\begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix} = -2$$

The other two matrices are totally unimodular.

22.3 Cutting Planes

The *cutting plane method* is an exact method for solving mixed-integer programs when $\mathbf{A}$ is not totally unimodular.[6] Modern practical methods for solving integer programs use *branch and cut* algorithms[7] that combine the cutting plane method with the branch and bound method, discussed in the next section. The cutting plane method works by solving the relaxed LP and then adding linear constraints that result in an optimal solution to the original problem.

[6] R. E. Gomory, "An Algorithm for Integer Solutions to Linear Programs," *Recent Advances in Mathematical Programming*, vol. 64, pp. 269–302, 1963.

[7] M. Padberg and G. Rinaldi, "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems," *SIAM Review*, vol. 33, no. 1, pp. 60–100, 1991.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
isint(x, ϵ=1e-10) = abs(round(x) - x) ≤ ϵ
function is_totally_unimodular(A::Matrix)
    # all entries must be in [0,1,-1]
    if any(a ∉ (0,-1,1) for a in A)
        return false
    end
    # brute force check every subdeterminant
    r,c = size(A)
    for i in 1 : min(r,c)
        for a in combinations(1:r, i)
            for b in combinations(1:c, i)
                B = A[a,b]
                if det(B) ∉ (0,-1,1)
                    return false
                end
            end
        end
    end
    return true
end
function is_totally_unimodular(MIP)
    return is_totally_unimodular(MIP.A) &&
           all(isint, MIP.b) && all(isint, MIP.c)
end
```

Algorithm 22.3. Methods for determining whether matrices `A` or mixed-integer programs `MIP` are totally unimodular. The method `isint` returns true if the given value is integral. A more efficient method for determining total unimodularity is provided by K. Truemper, "A Decomposition Theory for Matroids. V. Testing of Matrix Total Unimodularity," *Journal of Combinatorial Theory, Series B*, vol. 49, no. 2, pp. 241–281, 1990.

We begin the cutting plane method with a solution $\mathbf{x}_c^*$ to the relaxed LP that is a vertex of $\mathbf{A}\mathbf{x} = \mathbf{b}$. If the $\mathcal{D}$ components in $\mathbf{x}_c^*$ are integral, then it is also an optimal solution to the original mixed-integer program, and we are done. If the $\mathcal{D}$ components in $\mathbf{x}_c^*$ are not integral, we find a hyperplane with $\mathbf{x}_c^*$ on one side and all feasible discrete solutions on the other. This *cutting plane* is an additional linear constraint to exclude $\mathbf{x}_c^*$. The augmented LP is then solved for a new $\mathbf{x}_c^*$.

Each iteration of algorithm 22.4 introduces cutting planes that make nonintegral components of $\mathbf{x}_c^*$ infeasible while preserving the feasibility of the nearest integral solutions and the rest of the feasible set. The integer program modified with these cutting plane constraints is solved for a new relaxed solution. Figure 22.3 illustrates this process.

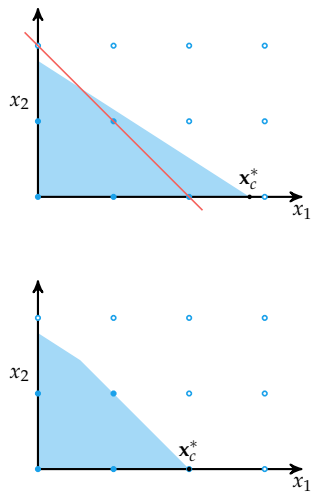


Figure 22.3. The cutting plane method introduces constraints until the solution to the LP is integral. The cutting plane is shown as a red line on the top plot. The feasible region of the augmented LP is on the bottom plot.

We wish to add constraints that cut out nonintegral components of $\mathbf{x}_c^*$. For an LP in equality form with constraint $\mathbf{A}\mathbf{x} = \mathbf{b}$, recall from section 12.2.1 that we can partition a vertex solution $\mathbf{x}_c^*$ to arrive at

$$\mathbf{A}_\mathcal{B}\mathbf{x}_\mathcal{B}^* + \mathbf{A}_\mathcal{V}\mathbf{x}_\mathcal{V}^* = \mathbf{b} \tag{22.4}$$

where $\mathbf{x}_\mathcal{V}^* = \mathbf{0}$. The nonintegral components of $\mathbf{x}_c^*$ will thus occur only in $\mathbf{x}_\mathcal{B}^*$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function cutting_plane(MIP)
    frac(x) = x - floor(x)
    LP = relax(MIP)
    x, b_inds = minimize_lp(LP)
    n_orig = length(x)
    v_inds = setdiff(1:n_orig, b_inds)
    D = copy(MIP.D)
    while !all(isint(x[i]) for i in D)
        AB, AV = LP.A[:,b_inds], LP.A[:,v_inds]
        Abar = AB\AV
        for i in copy(D)
            if !isint(x[i])
                b = findfirst(isequal(i), b_inds)
                A2 = [LP.A zeros(size(LP.A,1));
                      zeros(1,size(LP.A,2))+1)]
                A2[end,end] = 1
                A2[end,v_inds] = -frac.(Abar[b,:])
                b2 = [LP.b; -frac(x[i])]
                c2 = [LP.c; 0]
                LP = LinearProgram(A2,b2,c2)
                push!(D, size(LP.A, 2))
            end
        end
        x, b_inds = minimize_lp(LP)
        v_inds = setdiff(eachindex(x), b_inds)
    end
    return x[1:n_orig]
end
```

Algorithm 22.4. The cutting plane method solves a given mixed-integer program `MIP` and returns an optimal design vector. An error is thrown if no feasible solution exists. The helper function `frac` returns the fractional part of a number, and the implementation for `minimize_lp`, algorithm 12.5, has been adjusted to return the basic indices `b_inds` along with an optimal design `x`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We will now create a constraint for design index $b \in \mathcal{B} \cap \mathcal{D}$ where $\mathbf{x}_c^*$ is noninteger. Equation (22.4) holds for any feasible $\mathbf{x}$. Multiplying by $\mathbf{A}_\mathcal{B}^{-1}$ produces:

$$\mathbf{x}_\mathcal{B} + \mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_\mathcal{V}\mathbf{x}_\mathcal{V} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b} \tag{22.5}$$

or equivalently

$$\mathbf{x}_\mathcal{B} + \bar{\mathbf{A}}\mathbf{x}_\mathcal{V} = \bar{\mathbf{b}} \tag{22.6}$$

where $\bar{\mathbf{A}} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{A}_\mathcal{V}$ and $\bar{\mathbf{b}} = \mathbf{A}_\mathcal{B}^{-1}\mathbf{b}$.

We can expand equation (22.6) for component b into its integral and fractional parts:[8]

$$x_b + \sum_{v \in \mathcal{V}} \left(\bar{A}_{bv} + \lfloor \bar{A}_{bv} \rfloor - \lfloor \bar{A}_{bv} \rfloor\right) x_v = \bar{b}_b + \lfloor \bar{b}_b \rfloor - \lfloor \bar{b}_b \rfloor \tag{22.7}$$

and we move all integers to the left-hand side:[9]

$$x_b - \underbrace{\lfloor \bar{b}_b \rfloor}_{\text{integer}} + \sum_{v \in \mathcal{V}} \underbrace{\left(\lfloor \bar{A}_{bv} \rfloor\right)}_{\text{integer}} x_v = \underbrace{\bar{b}_b - \lfloor \bar{b}_b \rfloor}_{\text{fraction}} - \sum_{v \in \mathcal{V}} \underbrace{\left(\bar{A}_{bv} - \lfloor \bar{A}_{bv} \rfloor\right)}_{\text{fraction}} x_v \tag{22.8}$$

The right side contains the fractional terms. It must be less than 1, since $0 \leq \bar{b}_b - \lfloor \bar{b}_b \rfloor < 1$ and the remaining term is negative. The left side forces the right to be an integer, so:

$$\bar{b}_b - \lfloor \bar{b}_b \rfloor - \sum_{v \in \mathcal{V}} \left(\bar{A}_{bv} - \lfloor \bar{A}_{bv} \rfloor\right) x_v \leq 0 \tag{22.9}$$

will hold for all integers in the feasible set and will not hold for $\mathbf{x}_c^*$.

We can introduce this new cutting plane constraint to cut out our relaxed solution while preserving the feasibility of integer solutions in the feasible set. The constraint is written in equality form using a new integral slack variable x_k:

$$x_k + \sum_{v \in \mathcal{V}} \left(\lfloor \bar{A}_{bv} \rfloor - \bar{A}_{bv}\right) x_v = \lfloor \bar{b}_b \rfloor - \bar{b}_b \tag{22.10}$$

Using equation (22.5) with $\mathbf{x}_\mathcal{V}^* = \mathbf{0}$, we can replace $\bar{b}_b$ with x_b^* to get

$$x_k + \sum_{v \in \mathcal{V}} \left(\lfloor \bar{A}_{bv} \rfloor - \bar{A}_{bv}\right) x_v = \lfloor x_b^* \rfloor - x_b^* \tag{22.11}$$

Each iteration of algorithm 22.4 thus increases the number of constraints and the number of variables until solving the LP produces an integral solution. Only the components corresponding to the original design variables are returned.

The cutting plane method is used to solve a simple integer linear program in example 22.3.

[8] Note that $\lfloor x \rfloor$, or *floor* of x, rounds x down to the nearest integer.

[9] The floor of a number produces an integer, and the solution to our integer program requires that $\mathbf{x}$ be integral, so all entries on the left side will be integers, and the left side will evaluate to an integer.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 22.3. The cutting plane method used to solve an integer program.

Consider the integer program:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & 2x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & \begin{bmatrix} 0.5 & -0.5 & 1.0 \\ 2.0 & 0.5 & -1.5 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 2.5 \\ -1.5 \end{bmatrix} \\ & \mathbf{x} \geq \mathbf{0} \qquad \mathbf{x} \in \mathbb{Z}^3 \end{aligned}$$

The relaxed solution is $\mathbf{x}^* \approx [0.818, 0, 2.091]$, yielding:

$$\mathbf{A}_{\mathcal{B}} = \begin{bmatrix} 0.5 & 1 \\ 2 & -1.5 \end{bmatrix} \quad \mathbf{A}_{\mathcal{V}} = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \quad \bar{\mathbf{A}} = \begin{bmatrix} -0.091 \\ -0.455 \end{bmatrix} \quad \bar{\mathbf{b}} = \begin{bmatrix} 0.818 \\ 2.091 \end{bmatrix}$$

From equation (22.11), the constraint for x_1 with slack variable x_4 is:

$$\begin{aligned} x_4 + (\lfloor -0.091 \rfloor - (-0.091))x_2 &= \lfloor 0.818 \rfloor - 0.818 \\ x_4 - 0.909x_2 &= -0.818 \end{aligned}$$

The constraint for x_3 with slack variable x_5 is:

$$\begin{aligned} x_5 + (\lfloor -0.455 \rfloor - (-0.455))x_2 &= \lfloor 2.091 \rfloor - 2.091 \\ x_5 - 0.545x_2 &= -0.091 \end{aligned}$$

The modified integer program has:

$$\mathbf{A} = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 & 0 \\ 2 & 0.5 & -1.5 & 0 & 0 \\ 0 & -0.909 & 0 & 1 & 0 \\ 0 & -0.545 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 2.5 \\ -1.5 \\ -0.818 \\ -0.091 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \\ 0 \end{bmatrix}$$

Solving the modified LP, we get $\mathbf{x}^* \approx [0.9, 0.9, 2.5, 0.0, 0.4]$. Since this point is not integral, we repeat the procedure with constraints:

$$\begin{aligned} x_6 - 0.9x_4 &= -0.9 & \qquad x_7 - 0.9x_4 &= -0.9 \\ x_8 - 0.5x_4 &= -0.5 & \qquad x_9 - 0.4x_4 &= -0.4 \end{aligned}$$

and solve a third LP to obtain: $\mathbf{x}^* = [1, 2, 3, 1, 1, 0, 0, 0, 0]$ with a final solution of $\mathbf{x}_i^* = [1, 2, 3]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

22.4 Branch and Bound

One method for finding the global optimum of a discrete problem, such as an integer program, is to enumerate all possible solutions. The *branch and bound*[10] method guarantees that an optimal solution is found without having to evaluate all possible solutions. Many commercial integer program solvers use ideas from both the cutting plane method and branch and bound.[11] The method gets its name from the *branch operation* that partitions the solution space[12] and the *bound operation* that computes a lower bound for a partition.

Algorithm 22.5 uses a *priority queue*, which is a data structure that associates priorities with elements in a collection. We can add an element and its priority value to a priority queue using the *enqueue* operation. We can remove the element with the minimum priority value using the *dequeue* operation.

The algorithm begins with a priority queue containing a single LP relaxation of the original mixed-integer program. Associated with that LP is a solution $\mathbf{x}_c^*$ and objective value $y_c = \mathbf{c}^\top \mathbf{x}_c^*$. The objective value serves as a lower bound on the solution and thus serves as the LP's priority in the priority queue. At each iteration of the algorithm, we check whether the priority queue is empty. If it is not empty, we dequeue the LP with the lowest priority value. If the solution associated with that element has the necessary integral components, then we keep track of whether it is the best integral solution found so far.

If the dequeued solution has one or more components in $\mathcal{D}$ that are nonintegral, we choose from $\mathbf{x}_c^*$ a component that is farthest from an integer value. Suppose this component corresponds to the ith design variable. We *branch* by considering two new LPs, each one created by adding one of the following constraints to the dequeued LP:[13]

$$x_i \leq \lfloor x_{i,c}^* \rfloor \qquad \text{or} \qquad x_i \geq \lceil x_{i,c}^* \rceil \tag{22.12}$$

as shown in figure 22.4. Example 22.4 demonstrates this process.

We compute the solution associated with these two LPs, which provide lower bounds on the value of the original mixed-integer program. If either solution lowers the objective value when compared to the best integral solution seen so far, it is placed into the priority queue. Not placing solutions already known to be inferior to the best integral solution seen thus far allows branch and bound to prune the search space. The process continues until the priority queue is empty,

[10] Branch and bound is a general method that can be applied to many kinds of discrete optimization problems, but we will focus here on how it can be used for integer programming. A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[11] J. E. Mitchell, "Branch-And-Cut Algorithms for Combinatorial Optimization Problems," in *Handbook of Applied Optimization*, P. M. Pardalos and M. G. C. Resende, eds., Oxford University Press, 2002, pp. 65–77.

[12] The subsets are typically disjoint, but this is not required. For branch and bound to work, at least one subset must have an optimal solution. D. A. Bader, W. E. Hart, and C. A. Phillips, "Parallel Algorithm Design for Branch and Bound," in *Tutorials on Emerging Methodologies and Applications in Operations Research*, H. J. Greenberg, ed., Kluwer Academic Press, 2004.

[13] Note that $\lceil x \rceil$, or *ceiling* of x, rounds x up to the nearest integer.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function minimize_lp_and_y(LP)
    try
        x = minimize_lp(LP).x
        return (x, x⋅LP.c)
    catch
        return (fill(NaN, length(LP.c)), Inf)
    end
end
function branch_and_bound(MIP)
    LP = relax(MIP)
    x, y = minimize_lp_and_y(LP)
    best, n = (x=deepcopy(x), y=Inf), length(x)
    Q = PriorityQueue()
    enqueue!(Q, (LP,x,y), y)

    while !isempty(Q)
        LP, x, y = dequeue!(Q)
        if any(isnan.(x)) || all(isint(x[i]) for i in MIP.D)
            if y < best.y
                best = (x=x[1:n], y=y)
            end
        else
            i = argmax(j -> abs(x[j] - round(x[j])), MIP.D)
            # x_i ≤ floor(x_i)
            A, b, c = LP.A, LP.b, LP.c
            A2 = [A zeros(size(A,1));
                  [j==i for j in 1:size(A,2)]' 1]
            b2, c2 = [b; floor(x[i])], [c; 0]
            LP2 = LinearProgram(A2,b2,c2)
            x2, y2 = minimize_lp_and_y(LP2)
            if y2 ≤ best.y
                enqueue!(Q, (LP2,x2,y2), y2)
            end
            # x_i ≥ ceil(x_i)
            A2 = [A zeros(size(A,1));
                  [j==i for j in 1:size(A,2)]' -1]
            b2, c2 = [b; ceil(x[i])], [c; 0]
            LP2 = LinearProgram(A2,b2,c2)
            x2, y2 = minimize_lp_and_y(LP2)
            if y2 ≤ best.y
                enqueue!(Q, (LP2,x2,y2), y2)
            end
        end
    end
    return best.x
end
```

Algorithm 22.5. The branch and bound algorithm for solving a mixed-integer program `MIP`. The helper method solves an LP and returns both the solution and its value. An infeasible LP produces a `NaN` solution and an `Inf` value. More sophisticated implementations will drop variables whose solutions are known in order to speed computation. The `PriorityQueue` type is provided by the `DataStructures.jl` package.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Consider a relaxed solution $\mathbf{x}_c^* = [3, 2.4, 1.2, 5.8]$ for an integer program with $\mathbf{c} = [-1, -2, -3, -4]$. The lower bound is

$$y \geq \mathbf{c}^\top \mathbf{x}_c^* = -34.6$$

We branch on a nonintegral coordinate of $\mathbf{x}_c^*$, typically the one farthest from an integer value. In this case, we choose the first nonintegral coordinate, $x_{2,c}^*$, which is 0.4 from the nearest integer value. We then consider two new LPs, one with $x_2 \leq 2$ as an additional constraint and the other with $x_2 \geq 3$ as an additional constraint.

Example 22.4. An example of a single application of the branching step in branch and bound.

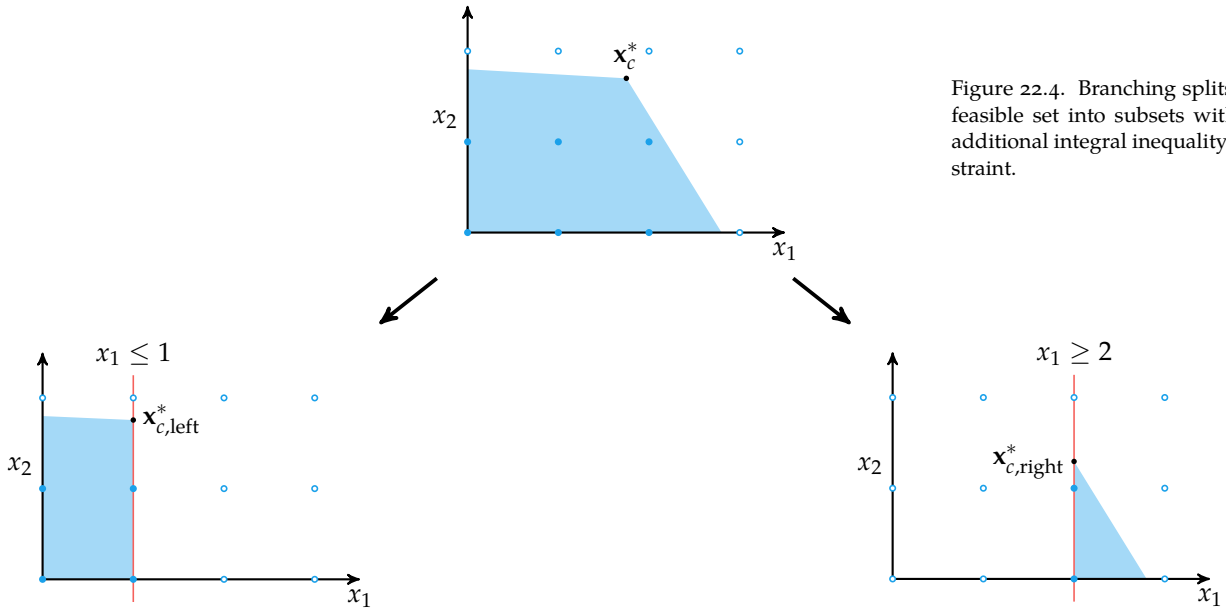


Figure 22.4. Branching splits the feasible set into subsets with an additional integral inequality constraint.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and we return the best feasible integral solution. Example 22.5 shows how branch and bound can be applied to a small integer program.

Example 22.5. Applying branch and bound to solve an integer programming problem.

We can use branch and bound to solve the integer program in example 22.3. As before, the relaxed solution is $\mathbf{x}_c^* = [0.818, 0, 2.091]$, with a value of 7.909. We branch on the first component, resulting in two integer programs, one with $x_1 \leq 0$ and one with $x_1 \geq 1$:

$$\mathbf{A}_{\text{left}} = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 \\ 2 & 0.5 & -1.5 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{b}_{\text{left}} = \begin{bmatrix} 2.5 \\ -1.5 \\ 0 \end{bmatrix} \quad \mathbf{c}_{\text{left}} = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

$$\mathbf{A}_{\text{right}} = \begin{bmatrix} 0.5 & -0.5 & 1 & 0 \\ 2 & 0.5 & -1.5 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix} \quad \mathbf{b}_{\text{right}} = \begin{bmatrix} 2.5 \\ -1.5 \\ 1 \end{bmatrix} \quad \mathbf{c}_{\text{right}} = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \end{bmatrix}$$

The left LP with $x_1 \leq 0$ is infeasible. The right LP with $x_1 \geq 1$ has a relaxed solution, $\mathbf{x}_c^* = [1, 2, 3, 0]$, and a value of 13. We have thus obtained our integral solution, $\mathbf{x}_i^* = [1, 2, 3]$.

22.5 *Dynamic Programming*

Dynamic programming[14] is a technique that can be applied to problems with *optimal substructure* and *overlapping subproblems*. A problem has optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. Figure 22.5 shows an example.

A problem with overlapping subproblems solved recursively will encounter the same subproblem many times. Instead of enumerating exponentially many potential solutions, dynamic programming either stores subproblem solutions, and thereby avoids having to recompute them, or recursively builds the optimal solution in a single pass. Problems with recurrence relations often have overlapping subproblems. Figure 22.6 shows an example.

[14] The term dynamic programming was chosen by Richard Bellman to reflect the time-varying aspect of the problems he applied it to and to avoid the sometimes negative connotations words like *research* and *mathematics* had. He wrote, "I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities." R. Bellman, *Eye of the Hurricane: An Autobiography*. World Scientific, 1984. p. 159.

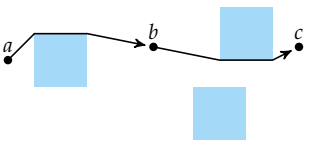


Figure 22.5. Shortest path problems have optimal substructure because if the shortest path from any a to c passes through b, then the subpaths $a \to b$ and $b \to c$ are both shortest paths. The blue boxes are obstacles that the path must avoid.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

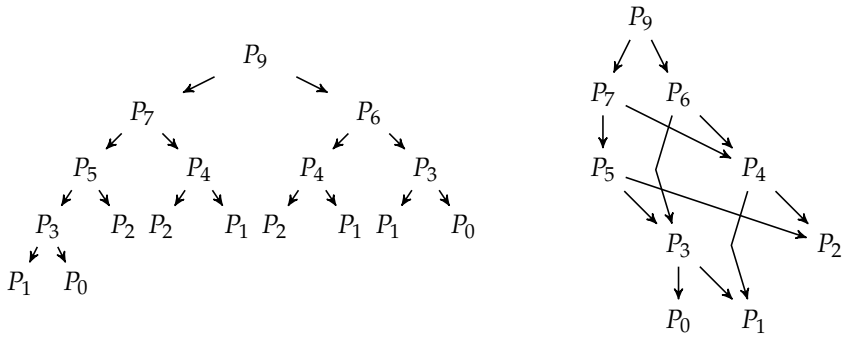


Figure 22.6. We can compute the nth term of the *Padovan sequence*, $P_n = P_{n-2} + P_{n-3}$, with $P_0 = P_1 = P_2 = 1$ by recursing through all subterms (left). A more efficient approach is to compute subterms once and reuse their values in subsequent calculations by exploiting the problem's overlapping subproblems (right).

Dynamic programming can be implemented either top-down or bottom-up, as demonstrated in algorithm 22.6. The top-down approach begins with the desired problem and recurses down to smaller and smaller subproblems. Subproblem solutions are stored so that when we are given a new subproblem, we can either retrieve the computed solution or solve and store it for future use.[15] The bottom-up approach starts by solving the smaller subproblems and uses their solutions to obtain solutions to larger problems.

[15] Storing subproblem solutions in this manner is called *memoization*.

```
function padovan_topdown(n, P=Dict())
    if !haskey(P, n)
        P[n] = n < 3 ? 1 :
               padovan_topdown(n-2,P) + padovan_topdown(n-3,P)
    end
    return P[n]
end
function padovan_bottomup(n)
    P = Dict(0⇒1,1⇒1,2⇒1)
    for i in 3 : n
        P[i] = P[i-2] + P[i-3]
    end
    return P[n]
end
```

Algorithm 22.6. Computing the Padovan sequence using dynamic programming, with both the top-down and bottom-up approaches.

The *knapsack problem* is a well-known combinatorial optimization problem that often arises in resource allocation.[16] Suppose we are packing our knapsack for a trip, but we have limited space and want to pack the most valuable items. There are several variations of the knapsack problem. In the 0-1 knapsack problem, we have n items, with the ith item having integral weight $w_i > 0$ and value

[16] The knapsack problem is an integer program with a single constraint, but it can be efficiently solved using dynamic programming.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

v_i. The design vector $\mathbf{x}$ consists of binary values that indicate whether an item is packed. The total weight cannot exceed our integral capacity $w_{\max}$, and we seek to maximize the total value of packed items. Hence, we have the following optimization problem:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & -\sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq w_{\max} \\ & x_i \in \{0, 1\} \text{ for } i \text{ in } 1:n \end{aligned} \tag{22.13}$$

There are 2^n possible design vectors, which makes direct enumeration for large n intractable. However, we can use dynamic programming. The 0-1 knapsack problem has optimal substructure and overlapping subproblems. Suppose we have already solved the knapsack problems involving the first $i-1$ items and all capacities up to $w_{\max}$. Now consider a larger problem with one additional item, item i with weight w_i and value v_i. The optimal solution will either include or exclude the new item.

- If it is not worth including the new item, the solution will have the same value as a knapsack with $i-1$ items and capacity $w_{\max}$.
- If it is worth including the new item, the solution will have the value of a knapsack with $i-1$ items and capacity $w_{\max} - w_i$ plus the value of the new item v_i.

We can define a recurrence relation $\text{knapsack}(i, w_{\max})$ that returns the sum of the values of the items in an optimally packed knapsack when considering the first i items with a remaining capacity of $w_{\max}$. This recurrence relation can be written as follows:

$$\text{knapsack}(i, w_{\max}) = \begin{cases} 0 & \text{if } i = 0 \\ \text{knapsack}(i-1, w_{\max}) & \text{if } w_i > w_{\max} \\ \max \begin{cases} \text{knapsack}(i-1, w_{\max}) & \text{(discard new item)} \\ \text{knapsack}(i-1, w_{\max} - w_i) + v_i & \text{(include new item)} \end{cases} & \text{otherwise} \end{cases} \tag{22.14}$$

Algorithm 22.7 provides an implementation.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function knapsack(v, w, w_max)
    n = length(v)
    y = Dict((0,j) ⇒ 0.0 for j in 0:w_max)
    for i in 1 : n
        for j in 0 : w_max
            y[i,j] = w[i] > j ? y[i-1,j] :
                     max(y[i-1,j],
                         y[i-1,j-w[i]] + v[i])
        end
    end
    # recover solution
    x, j = falses(n), w_max
    for i in n: -1 : 1
        if w[i] ≤ j && y[i,j] - y[i-1, j-w[i]] == v[i]
            # the ith element is in the knapsack
            x[i] = true
            j -= w[i]
        end
    end
    return x
end
```

Algorithm 22.7. A method for solving the 0-1 knapsack problem with item values `v`, integral item weights `w`, and integral capacity `w_max`. Recovering the design vector from the cached solutions requires additional iteration.

22.6 Ant Colony Optimization

Ant colony optimization[17] is a stochastic method for optimizing paths through graphs. This method was inspired by some ant species that wander randomly in search of food, leaving *pheromone* trails as they go. Other ants that stumble upon a pheromone trail are likely to start following it, thereby reinforcing the trail's scent. Pheromones slowly evaporate over time, causing unused trails to fade. Short paths, with stronger pheromones, are traveled more often and thus attract more ants. Thus, short paths create positive feedback that leads other ants to follow and further reinforce the shorter path.

Basic shortest path problems, such as the shortest paths found by ants between the ant hill and sources of food, can be efficiently solved using dynamic programming. Ant colony optimization has been used to find near-optimal solutions to the *traveling salesman problem*, a much more difficult problem in which we want to find the shortest path that passes through each node of the graph exactly once. Ant colony optimization has also been used to route multiple vehicles, find optimal locations for factories, and fold proteins.[18] The algorithm is stochastic in nature and is thus resistant to changes to the problem over time, such as traffic delays

[17] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[18] These and other applications are discussed in these references: M. Manfrin, "Ant Colony Optimization for the Vehicle Routing Problem," Ph.D. dissertation, Université Libre de Bruxelles, 2004. T. Stützle, "MAX-MIN Ant System for Quadratic Assignment Problems," Technical University Darmstadt, Tech. Rep., 1997. and A. Shmygelska, R. Aguirre-Hernández, and H. H. Hoos, "An Ant Colony Algorithm for the 2D HP Protein Folding Problem," in *International Workshop on Ant Algorithms (ANTS)*, 2002.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

changing effective edge lengths in the graph or networking issues that remove edges entirely.

Ants move stochastically based on the attractiveness of the edges available to them. The attractiveness of transition $i \to j$ depends on the pheromone level and an optional prior factor:

$$A(i \to j) = \tau(i \to j)^\alpha \eta(i \to j)^\beta \tag{22.15}$$

where α and β are exponents for the pheromone level τ and prior factor η, respectively.[19] For problems involving shortest paths, we can set the prior factor to the inverse edge length $\ell(i \to j)$ to encourage the traversal of shorter paths: $\eta(i \to j) = 1/\ell(i \to j)$. A method for computing the edge attractiveness is given in algorithm 22.8.

[19] Dorigo, Maniezzo, and Colorni recommend $\alpha = 1$ and $\beta = 5$.

Suppose an ant is at node i and can transition to any of the nodes $j \in \mathcal{J}$. The set of successor nodes $\mathcal{J}$ contains all valid outgoing neighbors.[20] Sometimes edges are excluded, such as in the traveling salesman problem where ants are prevented from visiting the same node twice. It follows that $\mathcal{J}$ is dependent on both i and the ant's history.

[20] The *outgoing neighbors* of a node i are all nodes j such that $i \to j$ is in the graph. In an undirected graph, the neighbors and the outgoing neighbors are identical.

```
function edge_attractiveness(G, τ, η; α=1, β=5)
    A = Dict()
    for i in 1 : nv(G)
        neighbors = outneighbors(G, i)
        for j in neighbors
            v = τ[(i,j)]^α * η[(i,j)]^β
            A[(i,j)] = v
        end
    end
    return A
end
```

Algorithm 22.8. A method for computing the edge attractiveness table given a graph `G`, pheromone levels `τ`, prior edge weights `η`, pheromone exponent `α`, and prior exponent `β`.

The probability of edge transition $i \to j$ is:

$$P(i \to j) = \frac{A(i \to j)}{\sum_{j' \in \mathcal{J}} A(i \to j')} \tag{22.16}$$

Ants affect subsequent generations by depositing pheromones. There are several methods for modeling pheromone deposition. One approach is to deposit pheromones after a path is completed.[21] Ants that do not find a path do not deposit pheromones. For shortest path problems, a successful ant that has established a path of length ℓ deposits $1/\ell$ pheromones on each edge it traversed.

[21] M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant Algorithms for Discrete Optimization," *Artificial Life*, vol. 5, no. 2, pp. 137–172, 1999.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function run_ant(G, lengths, τ, A, x_best, y_best)
    x = [1]
    while length(x) < nv(G)
        i = x[end]
        neighbors = setdiff(outneighbors(G, i), x)
        if isempty(neighbors) # ant got stuck
            return (x=x_best, y=y_best)
        end

        as = [A[(i,j)] for j in neighbors]
        push!(x, neighbors[sample(Weights(as))])
    end

    l = sum(lengths[(x[i-1],x[i])] for i in 2:length(x))
    for i in 2 : length(x)
        τ[(x[i-1],x[i])] += 1/l
    end
    if l < y_best
        return (x=x, y=l)
    else
        return (x=x_best, y=y_best)
    end
end
```

Algorithm 22.9. A method for simulating a single ant on a traveling salesman problem in which the ant starts at the first node and attempts to visit each node exactly once. Pheromone levels are increased at the end of a successful tour. The parameters are the graph `G`, edge lengths `lengths`, pheromone levels `τ`, edge attractiveness `A`, the best solution found thus far `x_best`, and its value `y_best`.

Ant colony optimization also models pheromone evaporation, which naturally occurs in the real world. Modeling evaporation helps prevent the algorithm from prematurely converging to a single, potentially suboptimal, solution. Pheromone evaporation is executed at the end of each iteration after all ant simulations have been completed. Evaporation decreases the pheromone level of each transition by a factor of $1 - \rho$, with $\rho \in [0, 1]$.[22]

[22] It is common to use $\rho = 1/2$.

For m ants at iteration k, the effective pheromone update is

$$\tau(i \to j)^{(k+1)} = (1 - \rho)\tau(i \to j)^{(k)} + \sum_{a=1}^{m} \frac{1}{\ell^{(a)}} \left((i \to j) \in \mathcal{P}^{(a)} \right) \tag{22.17}$$

where $\ell^{(a)}$ is the path length and $\mathcal{P}^{(a)}$ is the set of edges traversed by ant a.

Ant colony optimization is implemented in algorithm 22.10, with individual ant simulations using algorithm 22.9. Figure 22.7 shows ant colony optimization on a traveling salesman problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function ant_colony_optimization(G, lengths;
    m = 1000, k_max=100, α=1.0, β=5.0, ρ=0.5,
    η = Dict((e.src,e.dst)⇒1/lengths[(e.src,e.dst)]
             for e in edges(G)))
    τ = Dict((e.src,e.dst)⇒1.0 for e in edges(G))

    best = (x=[], y=Inf)
    for k in 1 : k_max
        A = edge_attractiveness(G, τ, η, α=α, β=β)
        for (e,v) in τ
            τ[e] = (1-ρ)*v
        end
        for ant in 1 : m
            best = run_ant(G,lengths,τ,A,best.x,best.y)
        end
    end
    return best.x
end
```

Algorithm 22.10. Ant colony optimization, which takes a directed or undirected graph `G` from `Graphs.jl` and a dictionary of edge tuples to edge lengths `lengths`. Ants start at the first node in the graph. Optional parameters include the number of ants per iteration `m`, the number of iterations `k_max`, the pheromone exponent `α`, the prior exponent `β`, the evaporation scalar `ρ`, and a dictionary of prior edge weights `η`.

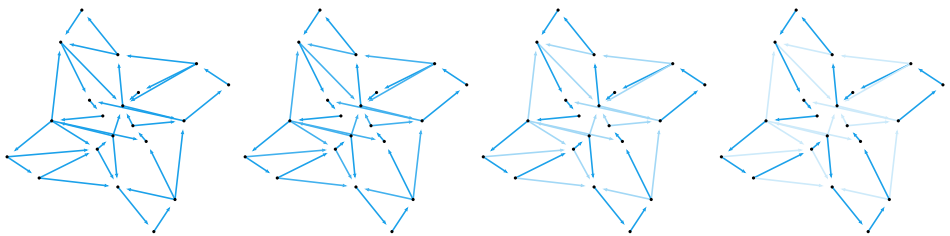


Figure 22.7. Ant colony optimization used to solve a traveling salesman problem on a directed graph using 50 ants per iteration. Path lengths are the Euclidean distances. Color opacity corresponds to pheromone level.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

22.7 Summary

- Discrete optimization problems require that the design variables be chosen from discrete sets.
- Relaxation, in which the continuous version of the discrete problem is solved, is by itself an unreliable technique for finding an optimal discrete solution but is central to more sophisticated algorithms.
- Many combinatorial optimization problems can be framed as an integer program, which is a linear program with integer constraints.
- Both the cutting plane and branch and bound methods can be used to solve integer programs efficiently and exactly. The branch and bound method is quite general and can be applied to a wide variety of discrete optimization problems.
- Dynamic programming is a powerful technique that exploits optimal substructure and overlapping subproblems in some problems.
- Ant colony optimization is a nature-inspired algorithm that can be used for optimizing paths in graphs.

22.8 Exercises

Exercise 22.1. A *Boolean satisfiability problem*, often abbreviated SAT, requires determining whether a Boolean design exists that causes a Boolean-valued objective function to output `true`. SAT problems were the first to be proven to belong to the difficult class of NP-complete problems.[23] This means that SAT is at least as difficult as all other problems whose solutions can be verified in polynomial time.

Consider the Boolean objective function:

$$f(\mathbf{x}) = x_1 \land (x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2)$$

Find an optimal design using enumeration. How many designs must be considered for an n-dimensional design vector in the worst case?

Solution: Enumeration tries all designs. Each component can either be true or false, thus resulting in 2^n possible designs in the worst case. This problem has $2^3 = 8$ possible designs.

[23] S. Cook, "The Complexity of Theorem-Proving Procedures," in *ACM Symposium on Theory of Computing*, 1971.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
f(x) = x[1] && (x[2] || !x[3]) && (!x[1] || !x[2])
using IterTools
for x in Iterators.product([true,false], [true,false], [true,false])
    if f(x)
       @show(x)
       break
    end
end
```

```
x = (true, false, false)
```

Exercise 22.2. Formulate the problem in exercise 22.1 as an integer linear program. Can any Boolean satisfiability problem be formulated as an integer linear program?

Solution: Because the Boolean satisfiability problem simply seeks a valid solution, we set $\mathbf{c}$ to zero in the objective. For the constraints, $\mathbf{x}$ is constrained to be nonnegative and integral as with all integer linear programs. In addition, we let 1 correspond to `true` and 0 correspond to `false` and introduce the constraint $\mathbf{x} \leq \mathbf{1}$.

The $\wedge$ ''and'' statements divide f into separate Boolean expressions, each of which must be true. We convert the expressions to linear constraints:

$$\begin{aligned} x_1 &\implies x_1 \geq 1 \\ x_2 \vee \neg x_3 &\implies x_2 + (1 - x_3) \geq 1 \\ \neg x_1 \vee \neg x_2 &\implies (1 - x_1) + (1 - x_2) \geq 1 \end{aligned}$$

where each expression must be satisfied (≥ 1) and a negated variable $\neg x_i$ is simply $1 - x_i$.

The resulting integer linear program is:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & 0 \\ \text{subject to} \quad & x_1 \geq 1 \\ & x_2 - x_3 \geq 0 \\ & -x_1 - x_2 \geq -1 \\ & \mathbf{x} \leq \mathbf{1} \\ & \mathbf{x} \in \mathbb{N}^3 \end{aligned}$$

This approach can be used to transform any Boolean satisfiability problem into an integer linear program.

Exercise 22.3. Why are we interested in totally unimodular matrices? Furthermore, why does every totally unimodular matrix contain only entries that are 0, 1, or -1?

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

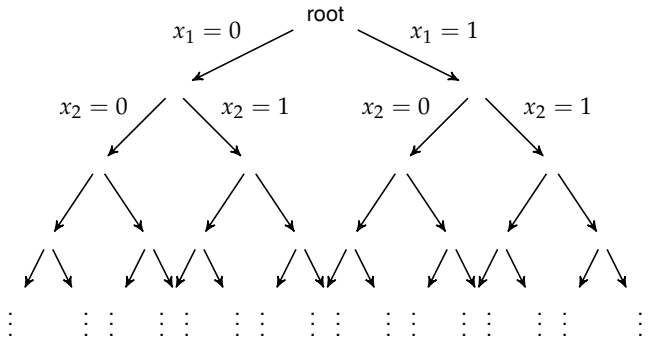
Solution: Integer programs for which $\mathbf{A}$ is totally unimodular and $\mathbf{b}$ is integral can be solved exactly using the simplex method. A matrix is totally unimodular if every square nonsingular submatrix is unimodular. A single matrix entry is a square submatrix. The determinant of a 1×1 matrix is its single entry. A single-entry submatrix is only unimodular if it has a determinant of ± 1, which occurs only for entries of ± 1. The single-entry submatrix can also be singular, which allows for 0. No other entries are permitted, and thus every totally unimodular matrix contains only entries that are 0, 1, or -1.

Exercise 22.4. This chapter solved the 0-1 knapsack problem using dynamic programming. Show how to apply branch and bound to the 0-1 knapsack problem, and use your approach to solve the knapsack problem with values $\mathbf{v} = [9, 4, 2, 3, 5, 3]$, and weights $\mathbf{w} = [7, 8, 4, 5, 9, 4]$ with capacity $w_{\max} = 20$.

Solution: The branch and bound method requires that we can perform the branching and bounding operations on our design.[24] The decisions being made in 0-1 knapsack are whether or not to include each item. Each item therefore represents a branch; either the item is included or it is excluded.

[24] P. J. Kolesar, "A Branch and Bound Algorithm for the Knapsack Problem," *Management Science*, vol. 13, no. 9, pp. 723–735, 1967.

A tree is constructed for every such enumeration according to:



Each node represents a subproblem in which certain items have already been included or excluded. The subproblem has the associated decision variables, values, and weights removed, and the capacity is effectively reduced by the total weight of the included items.

The branch and bound method avoids constructing the entire tree using information from the bounding operation. A bound can be constructed by solving a relaxed version of the knapsack subproblem. This *fractional knapsack* problem allows fractional values of items to be allocated, $0 \leq x_i \leq 1$.

The relaxed knapsack problem can be efficiently solved with a greedy approach. Items are added one at a time by selecting the next item with the greatest ratio of value to weight. If there is enough remaining capacity, the item is fully assigned with $x_i = 1$. If not, a fractional value is assigned such that the remaining capacity is saturated and all remaining items have $x_i = 0$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We begin by branching on the first item. The subtree with $x_1 = 0$ has the subproblem:

$$\begin{aligned} \underset{\mathbf{x}_{2:6}}{\text{minimize}} \quad & -\sum_{i=2}^{6} v_i x_i \\ \text{subject to} \quad & \sum_{i=2}^{6} w_i x_i \leq 20 \end{aligned}$$

whereas the subtree with $x_1 = 1$ has the subproblem:

$$\begin{aligned} \underset{\mathbf{x}_{2:6}}{\text{minimize}} \quad & -9 - \sum_{i=2}^{6} v_i x_i \\ \text{subject to} \quad & 7 + \sum_{i=2}^{6} w_i x_i \leq 20 \end{aligned}$$

We can construct a lower bound for both subtrees using the greedy approach. We sort the remaining items by value to weight:

item:	6	4	5	3	2
ratio:	3/4	3/5	5/9	2/4	4/8
	0.75	0.6	0.556	0.5	0.5

For the subtree with $x_1 = 0$, we fully allocate items 6, 4, and 5. We then partially allocate item 3 because we have remaining capacity 2, and thus set $x_3 = 2/4 = 0.5$. The lower bound is thus $-(3 + 3 + 5 + 2 \cdot 0.5) = -12$.

For the subtree with $x_1 = 1$, we allocate items 6 and 4 and partially allocate item 5 to $x_5 = 4/9$. The lower bound is thus $-9 - (3 + 3 + 5 \cdot (4/9)) \approx -17.222$.

The subtree with $x_1 = 1$ has the better lower bound, so the algorithm continues by splitting that subproblem. The final solution is $\mathbf{x} = [1, 0, 0, 0, 1, 1]$.

Exercise 22.5. Discrete variables are often used to control whether or not a constraint applies. Consider using a binary variable z to control whether one of two inequality constraints applies:

$$\begin{aligned} \underset{\mathbf{x},z}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & \mathbf{a}_0^\top \mathbf{x} \leq b_0 \quad \text{when } z = 0 \\ & \mathbf{a}_1^\top \mathbf{x} \leq b_1 \quad \text{when } z = 1 \\ & \mathbf{D}\mathbf{x} \leq \mathbf{c} \\ & \mathbf{x} \geq \mathbf{0} \\ & z \in \{0, 1\} \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We multiply by z or $(1 - z)$ to zero out the left-hand side when appropriate:

$$\begin{aligned}
\underset{\mathbf{x},z}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\
\text{subject to} \quad & z\,\mathbf{a}_0^\top \mathbf{x} \leq b_0 \\
& (1 - z)\,\mathbf{a}_1^\top \mathbf{x} \leq b_1 \\
& \mathbf{D}\mathbf{x} \leq \mathbf{c} \\
& \mathbf{x} \geq \mathbf{0} \\
& z \in \{0, 1\}
\end{aligned}$$

Unfortunately, these products prevent the overall optimization problem from being a mixed-integer linear program. Show that there is an alternative formulation by which the binary variable can be incorporated that results in a mixed-integer linear program. Assume that there is a large scalar value M that exceeds the maximum values of $\mathbf{a}_0^\top \mathbf{x} - b_0$ and $\mathbf{a}_1^\top \mathbf{x} - b_1$ for feasible values of $\mathbf{x}$.

Solution: We can use the large scalar M to ensure that each constraint is satisfied based on z:

$$\begin{aligned}
\underset{\mathbf{x},z}{\text{minimize}} \quad & \mathbf{c}^\top \mathbf{x} \\
\text{subject to} \quad & \mathbf{a}_0^\top \mathbf{x} \leq b_0 + Mz \\
& \mathbf{a}_1^\top \mathbf{x} \leq b_1 + M \cdot (1 - z) \\
& \mathbf{D}\mathbf{x} \leq \mathbf{c} \\
& \mathbf{x} \geq \mathbf{0} \\
& z \in \{0, 1\}
\end{aligned}$$

When $z = 0$, the first constraint behaves as normal and the second constraint is automatically satisfied, as M drives up the right-hand side. When $z = 1$, the opposite occurs. The added terms are linear in z, so the resulting constraints are linear, and the overall problem is a mixed-integer program. This technique is called the *big M method*.

Exercise 22.6. Consider a job shop scheduling problem with three jobs that each require completing four phases in a prescribed order on each of four different machines:

$$\begin{aligned}
&\text{job 1: machines } 1 \to 2 \to 3 \to 4 \\
&\text{job 2: machines } 2 \to 3 \to 4 \to 1 \\
&\text{job 3: machines } 3 \to 4 \to 1 \to 2
\end{aligned}$$

Each job phase has a duration:

$$\begin{aligned}
&\text{job 1: durations } [2, 3, 4, 1] \\
&\text{job 2: durations } [2, 2, 5, 3] \\
&\text{job 3: durations } [1, 2, 5, 3]
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The objective is to minimize the time T required to complete all jobs by optimizing the start time of each job phase and the binary precedences, where

- $t_{j,p}$ is the start time of phase p of job j.
- $z_{j,k,m}$ is a binary value that is 1 if job j precedes job $k \neq j$ on machine m.

Formulate this problem as a mixed-integer program, including T as a design variable, and enforce that:

- Job phase start times are nonnegative.
- Job phases must start after their previous phases end.
- If job j precedes job k on machine m, then job k must start after job j finishes on machine m.
- One job precedes the other for every job pair $j \neq k$.
- The total time to complete all jobs comes after the last job ends.

Solution: We can enforce that job phase start times are nonnegative with the constraint:

$$\mathbf{t} \geq \mathbf{0}$$

We can enforce that job phase p starts after the previous phase $p-1$ ends with:

$$t_{j,p} \geq t_{j,p-1} + \Delta t_{j,p-1}$$

where $\Delta t_{j,p}$ is the duration of phase p of job j. We include 9 such constraints for the 3 subsequent phase pairs for the 3 jobs.

We must enforce that job k starts after job j on machine m if job j precedes job k. These constraints are added for the specific job phases that occur on the specified machine. For example, for machine 1, we must ensure that job 1 phase 1 precedes job 2 phase 4 when $z_{1,2,1} = 1$. We formulate this using the big M method as:

$$t_{2,4} \geq t_{1,1} + \Delta t_{1,1} - M \times (1 - z_{1,2,1})$$

where M is a very large scalar value. The big M method effectively removes the constraint when job 1 phase 1 does not precede job 2 phase 4, while keeping the constraint linear. We include 24 such constraints for the 6 ordered job pairs across the 4 machines.

We can enforce that one job precedes the other in each job pair ($j \neq k$) and each machine m with

$$z_{j,k,m} + z_{k,j,m} = 1$$

We include 12 such constraints for the 3 unordered job pairs and 4 machines.

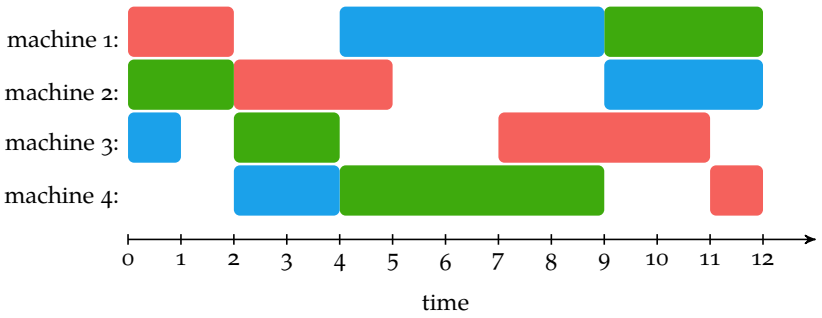
Finally, we enforce that the total time to complete all jobs T comes after the last job ends by enforcing that it comes after every job j:

$$T \geq t_{j,4} + \Delta t_{j,4}$$

We include 3 such constraints for the 3 jobs.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Exercise 22.7. Consider the following solution to the job shop scheduling problem from the previous question:



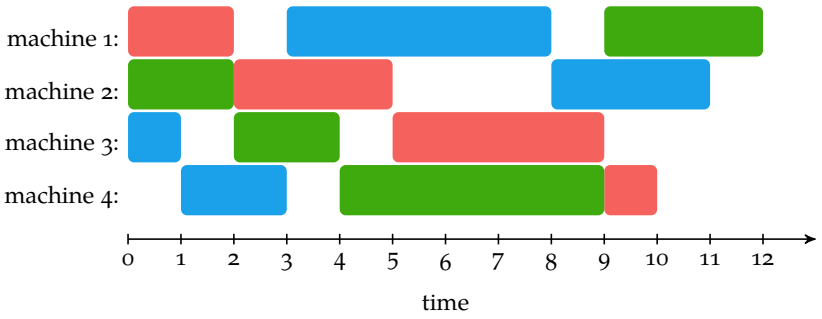
Job 1 is red, job 2 is green, and job 3 is blue. Although this is the optimal solution according to our objective, does this solution have any undesired properties? If yes, how could they be averted?

Solution: The solution above is an optimal solution output by a mixed-integer program solver that achieves the minimum total duration of 12. While this solution is optimal for our problem formulation, this schedule does not complete each job phase as soon as possible.

One way to remedy this issue is to solve the problem a second time starting from the current design, holding the precedences $\mathbf{z}$ fixed, but now minimizing all phase start times:

$$\text{minimize} \left(\sum_{j=1}^{3} \sum_{p=1}^{4} t_{j,p} \right)$$

Solving this second optimization problem yields:



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

23 Expression Optimization

Previous chapters discussed optimization over a fixed set of design variables. For many problems, the number of variables is unknown, such as in the optimization of graphical structures or computer programs.[1] Designs in these contexts can often be represented by expressions generated by a grammar. This chapter discusses ways to make the search of optimal designs more efficient by accounting for the grammatical structure of the design space.

23.1 Grammars

An expression can be represented by a tree of *symbols*. For example, the mathematical expression $x + \ln 2$ can be represented using the tree in figure 23.1 consisting of the symbols $+$, x, ln, and 2. *Grammars* specify the rules for generating valid expressions, thereby inducing constraints on the space of possible expressions.

A grammar is represented by a set of *production rules*. These rules involve symbols as well as *types*. A type can be interpreted as a set of expression trees. A production rule represents a possible expansion of a type into an expression involving symbols or types.[2] If a rule expands only to symbols, then it is called *terminal* because it cannot be expanded further. An example of a nonterminal rule is $\mathbb{R} \mapsto \mathbb{R} + \mathbb{R}$, which means that the type $\mathbb{R}$ can consist of elements of the set $\mathbb{R}$ added to elements in the set $\mathbb{R}$.[3]

We can generate an expression from a grammar by starting with a *start type* and then recursively applying different production rules. We stop when the tree contains only symbols. Figure 23.2 illustrates this process for the expression $x + \ln 2$. An application to natural language expressions is shown in example 23.1.

The number of possible expressions allowed by a grammar can be infinite. Example 23.2 shows a grammar that allows for infinitely many valid expressions.

[1] S. Gulwani, "Automating String Processing in Spreadsheets Using Input-Output Examples," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011. J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap, "Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 2, pp. 109–128, 1997.

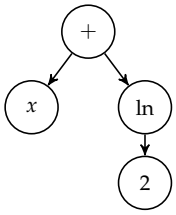


Figure 23.1. The expression $x + \ln 2$ represented as a tree.

[2] Types are sometimes referred to as *nonterminal symbols* (or simply *nonterminals*) because they can be expanded into a sequence of *terminal symbols* or *terminals* according to the production rules defining the grammar.

[3] This chapter focuses on *context-free grammars*, but other forms exist. See L. Kallmeyer, *Parsing Beyond Context-Free Grammars*. Springer, 2010.

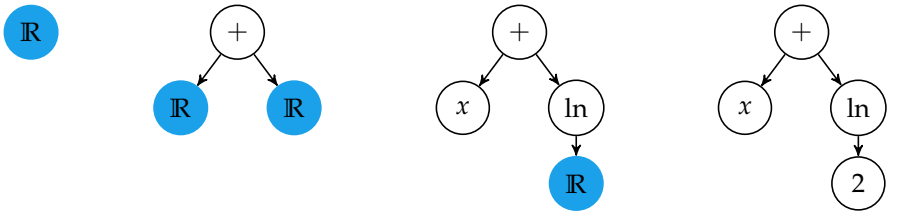


Figure 23.2. Using the production rules

$$\mathbb{R} \mapsto \mathbb{R} + \mathbb{R}$$
$$\mathbb{R} \mapsto x$$
$$\mathbb{R} \mapsto \ln(\mathbb{R})$$
$$\mathbb{R} \mapsto 2$$

to generate $x + \ln 2$. Blue nodes are unexpanded types.

Consider a grammar that allows for the generation of simple English statements:

$$\mathbb{S} \mapsto \mathbb{N}\,\mathbb{V}$$
$$\mathbb{V} \mapsto \mathbb{V}\,\mathbb{A}$$
$$\mathbb{A} \mapsto \text{rapidly} \mid \text{efficiently}$$
$$\mathbb{N} \mapsto \text{Alice} \mid \text{Bob} \mid \text{Mykel} \mid \text{Tim}$$
$$\mathbb{V} \mapsto \text{runs} \mid \text{reads} \mid \text{writes}$$

The types $\mathbb{S}$, $\mathbb{N}$, $\mathbb{V}$, and $\mathbb{A}$ correspond to statements, nouns, verbs, and adverbs, respectively. An expression is generated by starting with the type $\mathbb{S}$ and iteratively replacing types:

$$\mathbb{S}$$
$$\mathbb{N}\,\mathbb{V}$$
$$\text{Mykel}\ \mathbb{V}\,\mathbb{A}$$
$$\text{Mykel writes rapidly}$$

Not all terminal symbol categories must be used. For instance, the statement "Alice runs" can also be generated.

Example 23.1. A grammar for producing simple English statements. Using | on the right-hand side of an expression is shorthand for *or*. Thus, the rule

$$\mathbb{A} \mapsto \text{rapidly} \mid \text{efficiently}$$

is equivalent to having two rules, $\mathbb{A} \mapsto$ rapidly and $\mathbb{A} \mapsto$ efficiently.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Expression optimization often constrains expressions to a maximum depth or penalizes expressions based on their depth or node count. Even if the grammar allows a finite number of expressions, the space is often too vast to search exhaustively. Hence, there is a need for algorithms that efficiently search the space of possible expressions for one that optimizes an objective function.

Example 23.2. Some of the challenges associated with grammars, as illustrated with a four-function calculator grammar.

Consider a four-function calculator grammar that applies addition, subtraction, multiplication, and division to the ten digits:

$$\begin{aligned}
\mathbb{R} &\mapsto \mathbb{R} + \mathbb{R} \\
\mathbb{R} &\mapsto \mathbb{R} - \mathbb{R} \\
\mathbb{R} &\mapsto \mathbb{R} \times \mathbb{R} \\
\mathbb{R} &\mapsto \mathbb{R} \,/\, \mathbb{R} \\
\mathbb{R} &\mapsto 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

An infinite number of expressions can be generated because the nonterminal $\mathbb{R}$ can always be expanded into one of the calculator operations.

Many expressions will produce the same value. Addition and multiplication operators are commutative, meaning that the order does not matter. For example, $a + b$ is the same as $b + a$. These operations are also associative, meaning that grouping does not change the outcome of consecutive operations of the same type. For example, $(a \times b) \times c$ is the same as $a \times (b \times c)$. Other operations preserve values, like adding zero or multiplying by one.

Not all expressions under this grammar are mathematically valid. For example, division by zero is undefined. Removing zero as a terminal symbol is insufficient to prevent this error because zero can be constructed using other operations, such as $1 - 1$. Such exceptions are often handled by the objective function, which can catch exceptions and penalize them.

We can specify grammars in code as shown in example 23.3. Some basic operations on expression trees generated from a grammar are described in example 23.4. These operations are useful in the various algorithms described in this chapter.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We may define a grammar using the `grammar` macro. The nonterminals are on the left of the equal sign, and the expressions with terminals and nonterminals are on the right. The package includes some syntax to represent grammars more compactly.

```
using ExprRules
grammar = @grammar begin
    R = x          # reference a variable
    R = R * A      # multiple children
    R = f(R)       # call a function
    R = _(randn()) # random variable generated on node creation
    R = 1 | 2 | 3  # equivalent to R = 1, R = 2, or R = 3
    R = |(4:6)     # equivalent to R = 4, R = 5, or R = 6
    A = 7          # rules for different return types
end
```

Example 23.3. Example of defining a grammar using the `ExprRules.jl` package.

Many of the expression optimization algorithms involve manipulating components of an expression tree in a way that preserves the way the types were expanded. The `ExprRules.jl` package provides support for this. A `RuleNode` object represents a node in an expression tree. It includes links to child nodes, so it can be used to deduce the value of the subtree rooted at that node.

Calling `rand` with a specified starting type will generate a random expression represented by a `RuleNode`. Calling `sample` will select a random `RuleNode` from an existing `RuleNode` tree. Nodes are evaluated using `Core.eval`.

The method `return_type` returns the node's return type as a symbol, `isterminal` returns whether the symbol is terminal, `child_types` returns the list of nonterminal symbols associated with the node's production rule, and `nchildren` returns the number of children. These four methods each take as input the grammar and the node. The number of nodes in a `RuleNode` is obtained using `length(node)`, and the depth is obtained using `depth(node)`.

A `NodeLoc` type is used to refer to a node's location in the expression tree. Subtree manipulation often requires a `NodeLoc`, as shown below:

```
loc = sample(NodeLoc, node) # uniformly sample a node loc
loc = sample(NodeLoc, node, :R, grammar) # sample a node loc of type R
subtree = get(node, loc) # get the node at loc
```

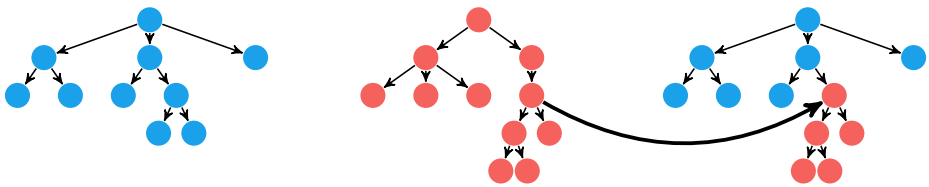
Example 23.4. Basic operations on expression trees generated from a grammar.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

23.2 *Genetic Programming*

Genetic algorithms (chapter 9) use chromosomes that encode design points in a sequential format. *Genetic programming*[4] represents individuals using trees instead (figure 23.3), which are better at representing mathematical functions, programs, decision trees, and other hierarchical structures. This book focuses only on genetic operations that adhere to the constraints of the grammar. Sometimes genetic programming with this restriction is referred to as *strongly typed genetic programming*.[5]

Similar to genetic algorithms, genetic programs are initialized randomly and support crossover and mutation. In *tree crossover* (figure 23.4), two parent trees are mixed to form a child tree. A random node is chosen in each parent, and the subtree at the chosen node in the first parent is replaced with the subtree at the chosen node of the second parent. Tree crossover works on parents with different sizes and shapes, allowing arbitrary trees to mix. In some cases one must ensure that replacement nodes have certain types, such as Boolean values input into the condition of an `if` statement. Tree crossover is implemented in algorithm 23.1.



Tree crossover tends to produce trees with greater depth than the parent trees. Each generation tends to increase in complexity, which often results in overly complicated solutions and slower runtimes. We encourage *parsimony*, or simplicity, in the solution, by introducing a small bias in the objective function value based on a tree's depth or node count.

Applying *tree mutation* (figure 23.5) starts by choosing a random node in the tree. The subtree rooted at that node is deleted, and a new random subtree is generated to replace the old subtree. In contrast to mutation in binary chromosomes, tree mutation can typically occur at most once, often with a low probability around 1%. Tree mutation is implemented in algorithm 23.2.

[4] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[5] D. J. Montana, "Strongly Typed Genetic Programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

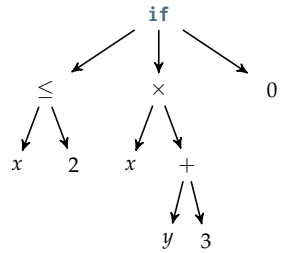


Figure 23.3. A tree representation of the Julia method: `x ≤ 2 ? x*(y+3) : 0`

Figure 23.4. Tree crossover is used to combine two parent trees to produce a child tree.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct TreeCrossover <: CrossoverMethod
    grammar   # rule set
    max_depth # maximum depth
end
function crossover(C::TreeCrossover, a, b)
    child = deepcopy(a)
    crosspoint = sample(b)
    typ = return_type(C.grammar, crosspoint.ind)
    d_subtree = depth(crosspoint)
    d_max = C.max_depth + 1 - d_subtree
    if d_max > 0 && contains_returntype(child, C.grammar, typ, d_max)
        loc = sample(NodeLoc, child, typ, C.grammar, d_max)
        insert!(child, loc, deepcopy(crosspoint))
    end
    return child
end
```

Algorithm 23.1. Tree crossover implemented for `a` and `b` of type `RuleNode` from `ExprRules.jl`. The `TreeCrossover` struct contains a rule set `grammar` and a maximum depth `max_depth`.

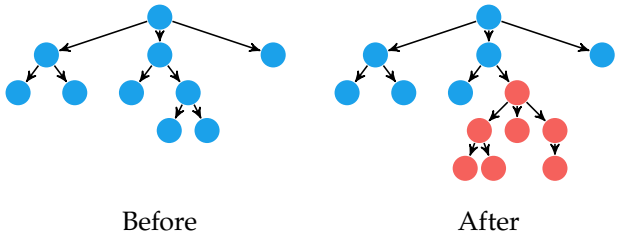


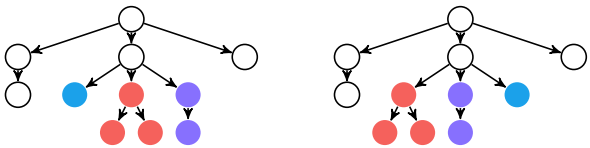
Figure 23.5. Tree mutation deletes a random subtree and generates a new one to replace it.

```
struct TreeMutation <: MutationMethod
    grammar # rule set
    p       # mutation probability
end
function mutate(M::TreeMutation, a)
    child = deepcopy(a)
    if rand() < M.p
        loc = sample(NodeLoc, child)
        typ = return_type(M.grammar, get(child, loc).ind)
        subtree = rand(RuleNode, M.grammar, typ)
        insert!(child, loc, subtree)
    end
    return child
end
```

Algorithm 23.2. Tree mutation implemented for an individual `a` of type `RuleNode` from `ExprRules.jl`. The `TreeMutation` struct contains a rule set `grammar` and a mutation probability `p`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Tree permutation (figure 23.6) is a second form of genetic mutation. The children of a randomly chosen node are randomly permuted. Tree permutation alone is typically not sufficient to introduce new genetic material and is often combined with tree mutation. Tree permutation is implemented in algorithm 23.3.



Before After

Figure 23.6. Tree permutation permutes the children of a randomly chosen node.

```
struct TreePermutation <: MutationMethod
    grammar # rule set
    p       # mutation probability
end
function mutate(M::TreePermutation, a)
    child = deepcopy(a)
    if rand() < M.p
        node = sample(child)
        n = length(node.children)
        types = child_types(M.grammar, node)
        for i in 1 : n-1
            c = 1
            for k in i+1 : n
                if types[k] == types[i] && rand() < 1/(c+=1)
                    node.children[i], node.children[k] =
                        node.children[k], node.children[i]
                end
            end
        end
    end
    return child
end
```

Algorithm 23.3. Tree permutation implemented for an individual `a` of type `RuleNode` from `ExprRules.jl`, where `p` is the mutation probability.

The implementation of genetic programming is otherwise identical to that of genetic algorithms. More care must typically be taken in implementing the crossover and mutation routines, particularly when determining what sorts of nodes can be generated and that only syntactically correct trees are produced. Genetic programming is used to generate an expression that approximates π in example 23.5.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 23.5. Using genetic programming to estimate π using only digits and the four principal arithmetic operations.

Consider approximating π using only operations on a four-function calculator. We can solve this problem using genetic programming where nodes can be any of the elementary operations: add, subtract, multiply, and divide, and the digits $1-9$.

We use `ExprRules.jl` to specify our grammar:

```
grammar = @grammar begin
    R = |(1:9)
    R = R + R
    R = R - R
    R = R / R
    R = R * R
end
```

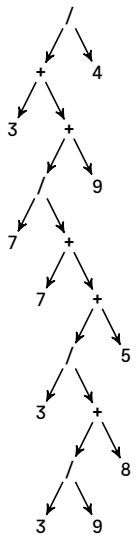
We construct an objective function and penalize large trees:

```
function f(node)
    value = Core.eval(node, grammar)
    if isinf(value) || isnan(value)
        return Inf
    end
    Δ = abs(value - π)
    return log(Δ) + length(node)/1e3
end
```

We finally run our genetic program using a call to the `population_method` function from section 9.2:

```
M = GeneticAlgorithm(TruncationSelection(50),
                     TreeCrossover(grammar, 10),
                     TreeMutation(grammar, 0.25))
population = [rand(RuleNode, grammar, :R) for i in 1:1000]
population = population_method(M, f, population, k_max)
best_tree = population[argmin(f.(population))]
```

The best performing tree is shown on the right. It evaluates to 3.141586, which matches π to four decimal places.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

23.3 Grammatical Evolution

Grammatical evolution[6] operates on an integer array instead of a tree, allowing the same techniques employed in genetic algorithms to be applied. Unlike genetic algorithms, the chromosomes in grammatical evolution encode expressions based on a grammar. Grammatical evolution was inspired by genetic material, which is inherently serial like the chromosomes used in genetic algorithms. [7]

In grammatical evolution, designs are integer arrays much like the chromosomes used in genetic algorithms. Each integer is unbounded because indexing is performed using modular arithmetic. The integer array can be translated into an expression tree by parsing it from left to right. We begin with a starting symbol and a grammar. Suppose n rules in the grammar can be applied to the starting symbol. The jth rule is applied, where $j = i \bmod_1 n$ and i is the first integer in the integer array.[8] We then consider the rules applicable to the resulting expression and use similar modular arithmetic based on the second integer in the array to select which rule to apply. This process is repeated until no rules can be applied and the phenotype is complete.[9] The decoding process is implemented in algorithm 23.4 and is worked through in example 23.6.

It is possible for the integer array to be too short, thereby causing the translation process to run past the length of the array. Rather than producing an invalid individual and penalizing it in the objective function, the process wraps around to the beginning of the array instead. This wrap-around effect means that the same decision can be read several times during the transcription process. Transcription can result in infinite loops, which can be prevented by a maximum depth.

Genetic operations work directly on the integer design array. We can adopt the operations used on real-valued chromosomes and apply them to the integer-valued chromosomes. The only change is that mutation must preserve integer values. A mutation method for integer-valued chromosomes using zero-mean Gaussian perturbations is implemented in algorithm 23.5.

Grammatical evolution uses two additional genetic operators. The first, *gene duplication*, occurs naturally as an error in DNA replication and repair. Gene duplication can allow new genetic material to be generated and can store a second copy of a useful gene to reduce the chance of a lethal mutation removing the gene from the gene pool. Gene duplication chooses a random interval of genes in the chromosome to duplicate. A copy of the selected interval is appended to the back of the chromosome. Duplication is implemented in algorithm 23.6.

[6] C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *European Conference on Genetic Programming*, 1998.

[7] Our serial DNA is read and used to construct complicated protein structures. DNA is often referred to as the *genotype*—the object on which genetic operations are performed. The protein structure is the *phenotype*—the object encoded by the genotype whose performance is evaluated. The grammatical evolution literature often refers to the integer design vector as the genotype and the resulting expression as the phenotype.

[8] We use $x \bmod_1 n$ to refer to the 1-index modulus:

$$((x-1) \mod n) + 1$$

This type of modulus is useful with 1-based indexing. The corresponding Julia function is `mod1`.

[9] No genetic information is read when there is only a single applicable rule.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct DecodedExpression
    node            # node in expression tree
    n_rules_applied # number of rules applied
end
function _decode(x, grammar, typ, c_max, c)
    types = grammar[typ]
    if length(types) > 1
        g = x[mod1(c+=1, length(x))]
        rule = types[mod1(g, length(types))]
    else
        rule = types[1]
    end
    node = RuleNode(rule)
    childtypes = child_types(grammar, node)
    if !isempty(childtypes) && c < c_max
        for ctyp in childtypes
            cnode, c = _decode(x, grammar, ctyp, c_max, c)
            push!(node.children, cnode)
        end
    end
    return (node, c)
end
function decode(x, grammar, sym, c_max=1000, c=0)
    node, c = _decode(x, grammar, sym, c_max, c)
    DecodedExpression(node, c)
end
```

Algorithm 23.4. A method for decoding an integer design vector to produce an expression, where `x` is a vector of integers, `grammar` is a `Grammar`, and `sym` is the root symbol. The counter `c` is used during the recursion process and the parameter `c_max` is an upper limit on the maximum number of rule applications, to prevent an infinite loop. The method returns a `DecodedExpression`, which contains the expression tree and the number of rules applied during the decoding process.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Consider a grammar for real-valued strings:

$$\begin{aligned}
\mathbb{R} &\mapsto \mathbb{D}\,\mathbb{D}'\,\mathbb{P}\,\mathbb{E} \\
\mathbb{D}' &\mapsto \mathbb{D}\,\mathbb{D}' \mid \epsilon \\
\mathbb{P} &\mapsto .\,\mathbb{D}\,\mathbb{D}' \mid \epsilon \\
\mathbb{E} &\mapsto \mathbb{E}\,\mathbb{S}\,\mathbb{D}\,\mathbb{D}' \mid \epsilon \\
\mathbb{S} &\mapsto + \mid - \mid \epsilon \\
\mathbb{D} &\mapsto 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

where $\mathbb{R}$ is a real value, $\mathbb{D}$ is a terminal decimal, $\mathbb{D}'$ is a nonterminal decimal, $\mathbb{P}$ is the decimal part, $\mathbb{E}$ is the exponent, and $\mathbb{S}$ is the sign. Any ϵ values produce empty strings.

Suppose our design is $[205, 52, 4, 27, 10, 59, 6]$ and we have the starting symbol $\mathbb{R}$. There is only one applicable rule, so we do not use any genetic information and we replace $\mathbb{R}$ with $\mathbb{D}\mathbb{D}'\mathbb{P}\mathbb{E}$.

Next we must replace $\mathbb{D}$. There are 10 options. We select $205 \bmod_1 10 = 5$, and thus obtain $4\mathbb{D}'\mathbb{P}\mathbb{E}$.

Next we replace $\mathbb{D}'$, which has two options. We select index $52 \bmod_1 2 = 2$, which corresponds to ϵ.

Continuing in this manner we produce the string *4E+8*.

The grammar can be implemented in `ExprRules` using:

```
grammar = @grammar begin
    R  =  D * De * P * E
    De  =  D * De | ""
    P  =  "." * D * De | ""
    E  =  "E" * S * D * De | ""
    S  =  "+"|"-"|""
    D  =  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
end
```

and can be evaluated using:

```
x = [205, 52, 4, 27, 10, 59, 6]
str = Core.eval(decode(x, grammar, :R).node, grammar)
```

Example 23.6. The process by which an integer design vector in grammatical evolution is decoded into an expression.

Our implementation is depth-first. If 52 were instead 51, the rule $\mathbb{D}' \mapsto \mathbb{D}\,\mathbb{D}'$ would be applied, followed by selecting a rule for the new $\mathbb{D}$, eventually resulting in *43950.950E+8*.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct IntegerGaussianMutation <: MutationMethod
    σ # standard deviation
end
function mutate(M::IntegerGaussianMutation, child)
    return child + round.(Int, randn(length(child)).*M.σ)
end
```

Algorithm 23.5. The Gaussian mutation method modified to preserve integer values for integer-valued chromosomes. Each value is perturbed by a zero-mean Gaussian random value with standard deviation σ and then rounded to the nearest integer.

```
struct GeneDuplication <: MutationMethod end
function mutate(M::GeneDuplication, child)
    n = length(child)
    i, j = rand(1:n), rand(1:n)
    interval = min(i,j) : max(i,j)
    return [child; deepcopy(child[interval])]
end
```

Algorithm 23.6. The gene duplication method used in grammatical evolution.

The second genetic operation, *pruning*, tackles a problem encountered during crossover. As illustrated in figure 23.7, crossover will select a crossover point at random in each chromosome and construct a new chromosome using the left side of the first and the right side of the second chromosome. Unlike genetic algorithms, the trailing entries in chromosomes of grammatical evolution may not be used; during parsing, once the tree is complete, the remaining entries are ignored. The more unused entries, the more likely it is that the crossover point lies in the inactive region, thus not providing new beneficial material. An individual is pruned with a specified probability, and, if pruned, its chromosome is truncated such that only active genes are retained. Pruning is implemented in algorithm 23.7.



Figure 23.7. Crossover applied to chromosomes in grammatical evolution may not affect the active genes in the front of the chromosome. The child shown here inherits all of the active shaded genes from parent a, so it will effectively act as an identical expression. Pruning was developed to overcome this issue.

```
struct GenePruning <: MutationMethod
    p # mutation probability
    grammar # rule set
    typ # type symbol
end
function mutate(M::GenePruning, child)
    if rand() < M.p
        c = decode(child, M.grammar, M.typ).n_rules_applied
        if c < length(child)
            child = child[1:c]
        end
    end
    return child
end
```

Algorithm 23.7. The gene pruning method used in grammatical evolution.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Like genetic programming, grammatical evolution can use the genetic algorithm method.[10] Algorithm 23.8 applies multiple mutation methods in order to use pruning, duplication, and standard mutation approaches.

[10] Genotype to phenotype mapping would occur in the objective function.

```
struct MultiMutate <: MutationMethod
    Ms # mutation methods
end
function mutate(M::MultiMutate, child)
    for m in M.Ms
        child = mutate(m, child)
    end
    return child
end
```

Algorithm 23.8. A method for applying all mutation methods stored in the vector `Ms`.

Grammatical evolution suffers from two primary drawbacks. First, it is difficult to tell whether the chromosome is feasible without decoding it into an expression. Second, a small change in the chromosome may produce a large change in the corresponding expression.

23.4 Probabilistic Grammars

A *probabilistic grammar*[11] adds a weight to each rule in a grammar. When sampling from all applicable rules for a given node, we select a rule stochastically according to the relative weights. The probability of an expression is the product of the probabilities of sampling each rule. Algorithm 23.9 implements the probability calculation. Example 23.7 demonstrates sampling an expression from a probabilistic grammar and computes its likelihood.

[11] T. L. Booth and R. A. Thompson, "Applying Probability Measures to Abstract Languages," *IEEE Transactions on Computers*, vol. C-22, no. 5, pp. 442–450, 1973.

```
struct ProbabilisticGrammar
    grammar # rule set
    ws      # mapping of types to weights
end
function probability(probgram, node)
    typ = return_type(probgram.grammar, node)
    i = findfirst(isequal(node.ind), probgram.grammar[typ])
    p = probgram.ws[typ][i] / sum(probgram.ws[typ])
    for c in node.children
        p *= probability(probgram, c)
    end
    return p
end
```

Algorithm 23.9. A method for computing the probability of an expression based on a probabilistic grammar, where `probgram` is a probabilistic grammar consisting of a grammar `grammar` and a mapping of types to weights for all applicable rules `ws`, and `node` is a `RuleNode` expression.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Optimization using a probabilistic grammar improves its weights with each iteration using elite samples from a population. At each iteration, a population of expressions is sampled and their objective function values are computed. Some number of the best expressions are considered the elite samples and can be used to update the weights. A new set of weights is generated for the probabilistic grammar, where the weight $w_i^{\mathbb{T}}$ for the ith production rule applicable to return type $\mathbb{T}$ is set to the number of times the production rule was used in generating the elite samples.[12] This update procedure is implemented in algorithm 23.10.

[12] Probabilistic grammars can be extended to more complicated probability distributions that consider other factors, such as the depth in the expression or local dependencies among siblings in subtrees. One approach is to use Bayesian networks. P. K. Wong, L. Y. Lo, M. L. Wong, and K. S. Leung, "Grammar-Based Genetic Programming with Bayesian Network," in *IEEE Congress on Evolutionary Computation (CEC)*, 2014.

23.5 Probabilistic Prototype Trees

The *probabilistic prototype tree*[13] is a different approach that learns a distribution for every node in the expression tree. Each node in a probabilistic prototype tree contains a probability vector representing a categorical distribution over the grammar's production rules. The probability vectors are updated to reflect knowledge gained from successive generations of expressions. The maximum number of children for a node is the maximum number of nonterminals among rules in the grammar.

Probability vectors are randomly initialized when a node is created. Random probability vectors can be drawn from a *Dirichlet distribution*, which is a distribution over discrete distributions.[14] The original implementation initializes terminals to a scalar value of 0.6 and nonterminals to 0.4. To handle grammars, we maintain a probability vector for applicable rules to each parent type. Algorithm 23.11 defines a node type and implements this initialization method.

[13] R. Salustowicz and J. Schmidhuber, "Probabilistic Incremental Program Evolution," *Evolutionary Computation*, vol. 5, no. 2, pp. 123–141, 1997.

[14] Named for the German mathematician Johann Peter Gustav Lejeune Dirichlet (1805–1859). D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

Expressions are sampled using the probability vectors in the probabilistic prototype tree. A rule in a node is drawn from the categorical distribution defined by the node's probability vector for the required return type, normalizing the associated probability vector values to obtain a valid probability distribution. The tree is traversed in depth-first order. This sampling procedure is implemented in algorithm 23.12 and visualized in figure 23.8.

Learning can use information either from an entire sampled population or from elite samples. Let the best expression in the current generation be x_{best} with value y_{best} and the best expression found so far be x_{elite} with value y_{elite}. The node probabilities are updated to increase the likelihood of generating x_{best}.[15]

[15] The original probabilistic prototype tree implementation will periodically increase the likelihood of generating x_{elite}.

The probability of generating x_{best} is the product of the probabilities of choosing each rule in x_{best} when traversing through the probabilistic prototype tree. We

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 23.7. Sampling an expression from a probabilistic grammar and computing the expression's likelihood.

Consider a probabilistic grammar for strings composed entirely of ''a''s:

$$\begin{aligned}
\mathbb{A} &\mapsto \text{a}\,\mathbb{A} & w_1^{\mathbb{A}} &= 1\\
&\mapsto \text{a}\,\mathbb{B}\,\text{a}\,\mathbb{A} & w_2^{\mathbb{A}} &= 3\\
&\mapsto \epsilon & w_3^{\mathbb{A}} &= 2\\
\mathbb{B} &\mapsto \text{a}\,\mathbb{B} & w_1^{\mathbb{B}} &= 4\\
&\mapsto \epsilon & w_2^{\mathbb{B}} &= 1
\end{aligned}$$

where we have a set of weights $\mathbf{w}$ for each parent type and ϵ is an empty string.

Suppose we generate an expression starting with the type $\mathbb{A}$. The probability distribution over the three possible rules is:

$$\begin{aligned}
P(\mathbb{A} \mapsto \text{a}\,\mathbb{A}) &= 1/(1+3+2) &&= 1/6\\
P(\mathbb{A} \mapsto \text{a}\,\mathbb{B}\,\text{a}\,\mathbb{A}) &= 3/(1+3+2) &&= 1/2\\
P(\mathbb{A} \mapsto \epsilon) &= 2/(1+3+2) &&= 1/3
\end{aligned}$$

Suppose we sample the second rule and obtain a $\mathbb{B}$ a $\mathbb{A}$.

Next we sample a rule to apply to $\mathbb{B}$. The probability distribution over the two possible rules is:

$$\begin{aligned}
P(\mathbb{B} \mapsto \text{a}\,\mathbb{B}) &= 4/(4+1) &&= 4/5\\
P(\mathbb{B} \mapsto \epsilon) &= 1/(4+1) &&= 1/5
\end{aligned}$$

Suppose we sample the second rule and obtain a ϵ a $\mathbb{A}$.

Next we sample a rule to apply to $\mathbb{A}$. Suppose we sample $\mathbb{A} \mapsto \epsilon$ to obtain a ϵ a ϵ, which produces the ''a''-string ''aa''. The probability of the sequence of rules applied to produce ''aa'' under the probabilistic grammar is:

$$P(\mathbb{A} \mapsto \text{a}\,\mathbb{B}\,\text{a}\,\mathbb{A})P(\mathbb{B} \mapsto \epsilon)P(\mathbb{A} \mapsto \epsilon) = \frac{1}{2} \cdot \frac{1}{5} \cdot \frac{1}{3} = \frac{1}{30}$$

Note that this is not the same as the probability of obtaining ''aa'', as other sequences of production rules could also have produced it.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function _update!(probgram, x)
    grammar = probgram.grammar
    typ = return_type(grammar, x)
    i = findfirst(isequal(x.ind), grammar[typ])
    probgram.ws[typ][i] += 1
    for c in x.children
        _update!(probgram, c)
    end
    return probgram
end
function update!(probgram, Xs)
    for w in values(probgram.ws)
        fill!(w,0)
    end
    for x in Xs
        _update!(probgram, x)
    end
    return probgram
end
```

Algorithm 23.10. A method for applying a learning update to a probabilistic grammar `probgram` based on an elite sample of expressions `Xs`.

```
struct PPTNode
    ps       # mapping of symbol to rule probabilities
    children # child PPTNodes
end
function PPTNode(grammar;
    w_terminal = 0.6,
    w_nonterm = 1-w_terminal)

    ps = Dict(typ => normalize!([isterminal(grammar, i) ?
                    w_terminal : w_nonterm
                    for i in grammar[typ]], 1)
              for typ in nonterminals(grammar))
    PPTNode(ps, PPTNode[])
end
function get_child(ppt::PPTNode, grammar, i)
    if i > length(ppt.children)
        push!(ppt.children, PPTNode(grammar))
    end
    return ppt.children[i]
end
```

Algorithm 23.11. A probabilistic prototype tree node type and associated initialization function where `ps` is a dictionary mapping a symbol corresponding to a return type to a probability vector over applicable rules, and `children` is a list of `PPTNode`s. The method `get_child` will automatically expand the tree when attempting to access a nonexistent child.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function rand(ppt, grammar, typ)
    rules = grammar[typ]
    rule_index = sample(rules, Weights(ppt.ps[typ]))
    ctypes = child_types(grammar, rule_index)

    arr = Vector{RuleNode}(undef, length(ctypes))
    node = iseval(grammar, rule_index) ?
        RuleNode(rule_index, Core.eval(grammar, rule_index), arr) :
        RuleNode(rule_index, arr)

    for (i,typ) in enumerate(ctypes)
        node.children[i] =
            rand(get_child(ppt, grammar, i), grammar, typ)
    end
    return node
end
```

Algorithm 23.12. A method for sampling an expression from a probabilistic prototype tree. The tree is expanded as needed.

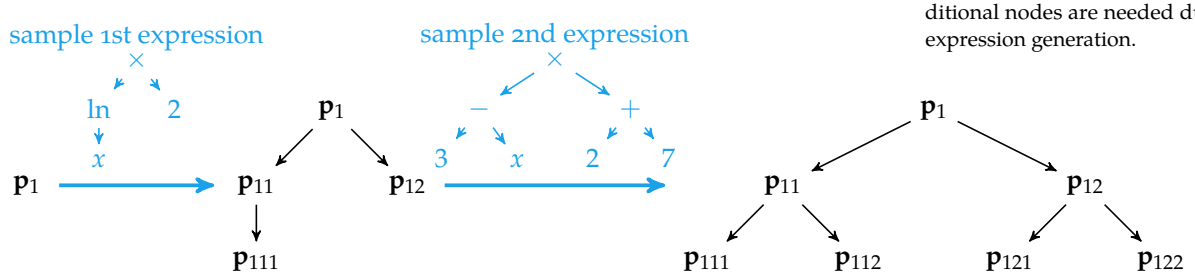


Figure 23.8. A probabilistic prototype tree initially contains only the root node but expands as additional nodes are needed during expression generation.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

compute a target probability for $P(x_{\text{best}})$:

$$P_{\text{target}} = P(x_{\text{best}}) + (1 - P(x_{\text{best}})) \cdot \alpha \cdot \frac{\epsilon - y_{\text{best}}}{\epsilon - y_{\text{elite}}} \tag{23.1}$$

where α and ϵ are positive constants. The fraction on the right-hand side produces larger steps toward expressions with better objective function values. The target probability can be calculated using algorithm 23.13.

```
function probability(ppt, grammar, expr)
    typ = return_type(grammar, expr)
    i = findfirst(isequal(expr.ind), grammar[typ])
    p = ppt.ps[typ][i]
    for (i,c) in enumerate(expr.children)
        p *= probability(get_child(ppt, grammar, i), grammar, c)
    end
    return p
end
function p_target(ppt, grammar, x_best, y_best, y_elite, α, ϵ)
    p_best = probability(ppt, grammar, x_best)
    return p_best + (1-p_best)*α*(ϵ - y_best)/(ϵ - y_elite)
end
```

Algorithm 23.13. Methods for computing the probability of an expression and the target probability, where `ppt` is the root node of the probabilistic prototype tree, `grammar` is the grammar, `expr` and `x_best` are `RuleNode` expressions, `y_best` and `y_elite` are scalar objective function values, and `α` and `ϵ` are scalar parameters.

The target probability is used to adjust the probability vectors in the probabilistic prototype tree. The probabilities associated with the chosen nodes are increased iteratively until the target probability is exceeded:

$$P\left(x_{\text{best}}^{(i)}\right) \leftarrow P\left(x_{\text{best}}^{(i)}\right) + c \cdot \alpha \cdot \left(1 - P\left(x_{\text{best}}^{(i)}\right)\right) \tag{23.2}$$

for all i where $x_{\text{best}}^{(i)}$ is the ith rule applied in expression x_{best} and c is a scalar.[16]

[16] The original paper recommends $c = 0.1$.

The adapted probability vectors are then renormalized to 1 by downscaling the values of all nonincreased vector components proportionally to their current value. The probability vector $\mathbf{p}$, where the ith component was increased, is adjusted according to:

$$p_j \leftarrow p_j \frac{1 - p_i}{\|\mathbf{p}\|_1 - p_i} \text{ for } j \neq i \tag{23.3}$$

The learning update is implemented in algorithm 23.14.

In addition to population-based learning, probabilistic prototype trees can also explore the design space via mutations. The tree is mutated to explore the region around $\mathbf{x}_{\text{best}}$. Let $\mathbf{p}$ be a probability vector in a node that was accessed when

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function _update!(ppt, grammar, x, c, α)
    typ = return_type(grammar, x)
    i = findfirst(isequal(x.ind), grammar[typ])
    p = ppt.ps[typ]
    p[i] += c*α*(1-p[i])
    psum = sum(p)
    for j in eachindex(p)
        if j != i
            p[j] *= (1- p[i])/(psum - p[i])
        end
    end
    for (pptchild,xchild) in zip(ppt.children, x.children)
        _update!(pptchild, grammar, xchild, c, α)
    end
    return ppt
end
function update!(ppt, grammar, x_best, y_best, y_elite, α, c, ϵ)
    p_targ = p_target(ppt, grammar, x_best, y_best, y_elite, α, ϵ)
    while probability(ppt, grammar, x_best) < p_targ
        _update!(ppt, grammar, x_best, c, α)
    end
    return ppt
end
```

Algorithm 23.14. A method for applying a learning update to a probabilistic prototype tree with root `ppt`, grammar `grammar`, best expression `x_best` with objective function value `y_best`, elite objective function value `y_elite`, step factor `α`, step factor multiplier `c`, and parameter `ϵ`.

generating $\mathbf{x}_{\text{best}}$. Each component in $\mathbf{p}$ is mutated with a probability inversely proportional to the problem size:

$$\frac{p_{\text{mutation}}}{\#\mathbf{p}\sqrt{\#\mathbf{x}_{\text{best}}}} \tag{23.4}$$

where p_{mutation} is a mutation parameter, $\#\mathbf{p}$ is the number of components in $\mathbf{p}$, and $\#\mathbf{x}_{\text{best}}$ is the number of rules applied in $\mathbf{x}_{\text{best}}$. A component i, selected for mutation, is adjusted according to:

$$p_i \leftarrow p_i + \beta \cdot (1 - p_i) \tag{23.5}$$

where β controls the amount of mutation. Small probabilities undergo larger mutations than do larger probabilities. All mutated probability vectors must be renormalized. Mutation is implemented in algorithm 23.15 and visualized in figure 23.9.

Finally, subtrees in the probabilistic prototype tree are pruned in order to remove stale portions of the tree. A child node is removed if its parent contains a probability component above a specified threshold that, when chosen, causes the child to be irrelevant. This is always the case for a terminal and may be the case

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
function mutate!(ppt, grammar, x_best, p_mutation, β;
    sqrtlen = sqrt(length(x_best)),
    )
    typ = return_type(grammar, x_best)
    p = ppt.ps[typ]
    prob = p_mutation/(length(p)*sqrtlen)
    for i in eachindex(p)
        if rand() < prob
            p[i] += β*(1-p[i])
        end
    end
    normalize!(p, 1)
    for (pptchild,xchild) in zip(ppt.children, x_best.children)
        mutate!(pptchild, grammar, xchild, p_mutation, β,
                sqrtlen=sqrtlen)
    end
    return ppt
end
```

Algorithm 23.15. A method for mutating a probabilistic prototype tree with root `ppt`, grammar `grammar`, best expression `x_best`, mutation parameter `p_mutation`, and mutation rate `β`.

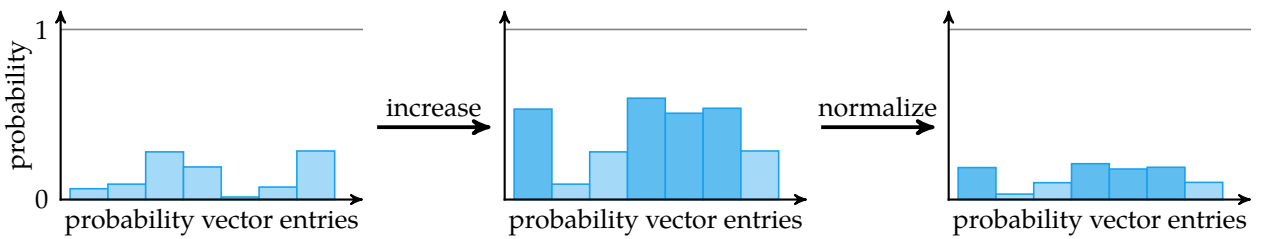


Figure 23.9. Mutating a probability vector in a probabilistic prototype tree with $\beta = 0.5$. The mutated components were increased according to equation (23.5), and the resulting probability vector was renormalized. Smaller probabilities receive greater increases.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

for a nonterminal. Pruning is implemented in algorithm 23.16 and demonstrated in example 23.8.

```
function prune!(ppt, grammar; p_threshold=0.99)
    kmax, pmax = :None, 0.0
    for (k, p) in ppt.ps
        pmax' = maximum(p)
        if pmax' > pmax
            kmax, pmax = k, pmax'
        end
    end
    if pmax > p_threshold
        i_successor_rule = argmax(ppt.ps[kmax])
        i_rule = grammar[kmax][i_successor_rule];
        if isterminal(grammar, i_rule)
            empty!(ppt.children)
        else
            max_arity_for_rule = maximum(nchildren(grammar, r) for
                                         r in grammar[kmax])
            while length(ppt.children) > max_arity_for_rule
                pop!(ppt.children)
            end
        end
    end
    for child in ppt.children
        prune!(child, grammar, p_threshold=p_threshold)
    end
    return ppt
end
```

Algorithm 23.16. A method for pruning a probabilistic prototype tree with root `ppt`, grammar `grammar`, and pruning probability threshold `p_threshold`.

23.6 Summary

- Expression optimization allows for optimizing tree structures that, under a grammar, can express sophisticated programs, structures, and other designs lacking a fixed size.
- Grammars define the rules used to construct expressions.
- Genetic programming adapts genetic algorithms to perform mutation and crossover on expression trees.
- Grammatical evolution operates on an integer array that can be decoded into an expression tree.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 23.8. An example of when pruning for probabilistic prototype trees should be applied.

Consider a node with a probability vector over the rule set:

$$\begin{aligned}\mathbb{R} &\mapsto \mathbb{R} + \mathbb{R}\\ \mathbb{R} &\mapsto \ln(\mathbb{R})\\ \mathbb{R} &\mapsto 2 \mid x\\ \mathbb{R} &\mapsto \mathbb{S}\end{aligned}$$

If the probability of selecting 2 or x grows large, then any children in the probabilistic prototype tree are unlikely to be needed and can be pruned. Similarly, if the probability of choosing $\mathbb{S}$ grows large, any children with return type $\mathbb{R}$ are unneeded and can be pruned.

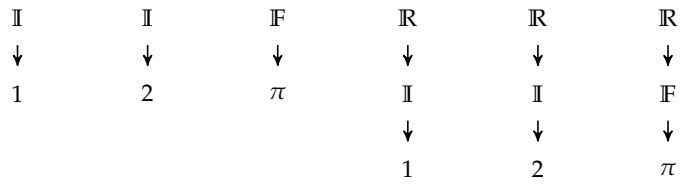
- Probabilistic grammars learn which rules are best to generate, and probabilistic prototype trees learn probabilities for every iteration of the expression rule generation process.

23.7 Exercises

Exercise 23.1. How many expression trees can be generated using the following grammar and the starting set $\{\mathbb{R}, \mathbb{I}, \mathbb{F}\}$?

$$\begin{aligned}\mathbb{R} &\mapsto \mathbb{I} \mid \mathbb{F}\\ \mathbb{I} &\mapsto 1 \mid 2\\ \mathbb{F} &\mapsto \pi\end{aligned}$$

Solution: Six expression trees can be generated:



Exercise 23.2. The number of expression trees up to height h that can be generated under a grammar grows super-exponentially. As a reference, calculate the number of expressions of height h that can be generated using the grammar:[17]

$$\mathbb{N} \mapsto \{\mathbb{N}, \mathbb{N}\} \mid \{\mathbb{N}, \} \mid \{, \mathbb{N}\} \mid \{\}$$

[17] Let an empty expression have height 0, the expression $\{\}$ have height 1, and so on.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: Only one expression of height 0 exists, and that is the empty expression. Let us denote this as $a_0 = 1$. Similarly, only one expression of height 1 exists, and that is the expression $\{\}$. Let us denote this as $a_1 = 1$. Three expressions exist for height 2, 21 for height 3, and so on.

Suppose we have constructed all expressions up to height h. All expressions of height $h+1$ can be constructed using a root node with left and right sub-expressions selected according to:

1. A left expression of height h and a right expression of height less than h
2. A right expression of height h and a left expression of height less than h
3. Left and right expressions of height h

It follows that the number of expressions of height $h+1$ is:[18]

$$a_{h+1} = 2a_h(a_0 + \cdots + a_{h-1}) + a_h^2$$

[18] This corresponds to OEIS sequence A001699.

Exercise 23.3. Define a grammar that can generate any nonnegative integer.

Solution: One can use the following grammar and the starting symbol $\mathbb{I}$:

$$\mathbb{I} \mapsto \mathbb{D} \mid \mathbb{D} + 10 \times \mathbb{I}$$
$$\mathbb{D} \mapsto 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Exercise 23.4. How do expression optimization methods handle divide-by-zero values or other exceptions encountered when generating random subtrees?

Solution: Constructing exception-free grammars can be challenging. Such issues can be avoided by catching the exceptions in the objective function and suitably penalizing them.

Exercise 23.5. Consider an arithmetic grammar such as:

$$\mathbb{R} \mapsto x \mid y \mid z \mid \mathbb{R} + \mathbb{R} \mid \mathbb{R} - \mathbb{R} \mid \mathbb{R} \times \mathbb{R} \mid \mathbb{R}/\mathbb{R} \mid \ln \mathbb{R} \mid \sin \mathbb{R} \mid \cos \mathbb{R}$$

Suppose the variables x, y, and z each have units, and the output is expected to be in particular units. How might such a grammar be modified to respect units?

Solution: There are many reasons why one must constrain the types of the variables manipulated during expression optimization. Many operators are valid only on certain inputs,[19] and matrix multiplication requires that the dimensions of the inputs be compatible. Physical dimensionality of the variables is another concern. The grammar must reason about the units of the input values and of the valid operations that can be performed on them.

[19] One typically does not take the square root of a negative number.

For instance, $x \times y$, with x having units $\text{kg}^a\text{m}^b\text{s}^c$, and y having units $\text{kg}^d\text{m}^e\text{s}^f$, will produce a value with units $\text{kg}^{a+d}\text{m}^{b+e}\text{s}^{c+f}$. Taking the square root of x will produce a value with units $\text{kg}^{a/2}\text{m}^{b/2}\text{s}^{c/2}$. Furthermore, operations such as sin can be applied only to unitless inputs.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

One approach for handling physical units is to associate an n-tuple with each node in the expression tree. The tuple records the exponent with respect to the allowable elementary units, which are specified by the user. If the elementary units involved are mass, length, and time, then each node would have a 3-tuple (a, b, c) to represent units $\text{kg}^a\text{m}^b\text{s}^c$. The associated grammar must take these units into account when assigning production rules.[20]

[20] For an overview, see A. Ratle and M. Sebag, "Genetic Programming and Domain Knowledge: Beyond the Limitations of Grammar-Guided Machine Discovery," in *International Conference on Parallel Problem Solving from Nature*, 2000.

Exercise 23.6. Consider the grammar

$$\begin{aligned}
\mathbb{S} &\mapsto \mathbb{NP}\ \mathbb{VP} \\
\mathbb{NP} &\mapsto \mathbb{ADJ}\ \mathbb{NP} \mid \mathbb{ADJ}\ \mathbb{N} \\
\mathbb{VP} &\mapsto \mathbb{V}\ \mathbb{ADV} \\
\mathbb{ADJ} &\mapsto \text{a} \mid \text{the} \mid \text{big} \mid \text{little} \mid \text{blue} \mid \text{red} \\
\mathbb{N} &\mapsto \text{mouse} \mid \text{cat} \mid \text{dog} \mid \text{pony} \\
\mathbb{V} &\mapsto \text{ran} \mid \text{sat} \mid \text{slept} \mid \text{ate} \\
\mathbb{ADV} &\mapsto \text{quietly} \mid \text{quickly} \mid \text{soundly} \mid \text{happily}
\end{aligned}$$

What is the phenotype corresponding to the genotype $[2, 10, 19, 0, 6]$ and the starting symbol $\mathbb{S}$?

Solution: The grammar can be encoded using `ExprRules.jl` using string composition:

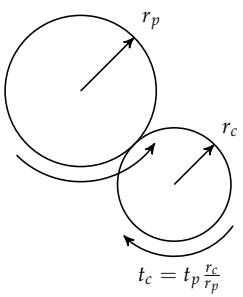
```
grammar = @grammar begin
    S = NP * " " * VP
    NP = ADJ * " " * NP
    NP = ADJ * " " * N
    VP = V  * " " * ADV
    ADJ = |(["a", "the", "big", "little", "blue", "red"])
    N = |(["mouse", "cat", "dog", "pony"])
    V = |(["ran", "sat", "slept", "ate"])
    ADV = |(["quietly", "quickly", "soundly", "happily"])
end
```

We can use our decode method to obtain the solution.

```
Core.eval(decode([2,10,19,0,6], grammar, :S)[1], grammar)
```

This produces ''little dog ate quickly''.

Exercise 23.7. Use genetic programming to evolve the gear ratios for a clock. Assume all gears are restricted to have radii selected from $\mathcal{R} = \{10, 25, 30, 50, 60, 100\}$. Each gear can either be attached to its parent's axle, thereby sharing the same rotation period, or be interlocked on its parent's rim, thereby having a rotation period depending on the parent's rotation period and on the gear ratio as shown to the right.



The clock can also contain hands, which are mounted on the axle of a parent gear. Assume the root gear turns with a period of $t_{\text{root}} = 0.1\,\text{s}$ and has a radius of 25. The objective is to produce a clock with a second, minute, and hour hand.

Score each individual according to:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

$$\left(\underset{\text{hands}}{\text{minimize}}\ \frac{(1-t_{\text{hand}})^2}{1^2}\right) + \left(\underset{\text{hands}}{\text{minimize}}\ (60-t_{\text{hand}})^2 60^2\right) + \left(\underset{\text{hands}}{\text{minimize}}\ (3600-t_{\text{hand}})^2 3600^2\right) + \#\text{nodes} \cdot 10^{-3}$$

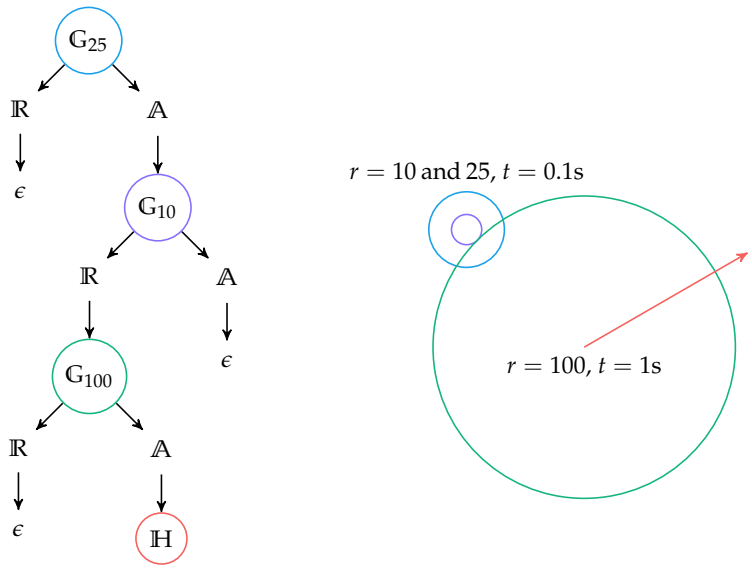
where t_{hand} is the rotation period of a particular hand in seconds and #nodes is the number of nodes in the expression tree. Ignore rotation direction.

Solution: We define a grammar for the clock problem. Let $\mathbb{G}_r$ be the symbol for a gear of radius r, let $\mathbb{A}$ be an axle, let $\mathbb{R}$ be a rim, and let $\mathbb{H}$ be a hand. Our grammar is:

$$\begin{aligned}\mathbb{G}_r &\mapsto \mathbb{R}\,\mathbb{A} \mid \epsilon\\ \mathbb{R} &\mapsto \mathbb{R}\,\mathbb{R} \mid \mathbb{G}_r \mid \epsilon\\ \mathbb{A} &\mapsto \mathbb{A}\,\mathbb{A} \mid \mathbb{G}_r \mid \mathbb{H} \mid \epsilon\end{aligned}$$

which allows each gear to have any number of rim and axle children. The expression ϵ is an empty terminal.

A clock with a single second hand can be constructed according to:



Note that the grammar does not compute the rotation period of the gears. This is handled by the objective function. A recursive procedure can be written to return the rotation periods of all hands. The list of rotation periods is subsequently used to compute the objective function value.

Exercise 23.8. The four 4s puzzle[21] is a mathematical challenge in which we use four 4 digits and mathematical operations to generate expressions for each of the integers from 0 to 100. For example, the first two integers can be produced by $4+4-4-4$ and $44/44$, respectively. Complete the four 4s puzzle.

[21] W. W. R. Ball, *Mathematical Recreations and Essays*. Macmillan, 1892.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: Any of the methods covered in this chapter can be used to complete the four 4s puzzle. A simple approach is to use Monte Carlo sampling on a suitable grammar. Sampled expressions with exactly four 4s are evaluated and, if suitable, are recorded. This procedure is repeated until an expression has been found for each integer.

One such suitable grammar is:[22]

$$\begin{aligned}\mathbb{R} \mapsto 4 \mid 44 \mid 444 \mid 4444 \mid \mathbb{R} + \mathbb{R} \mid \mathbb{R} - \mathbb{R} \mid \mathbb{R} \times \mathbb{R} \mid \mathbb{R} / \mathbb{R} \mid \\ \mathbb{R}^{\mathbb{R}} \mid \lfloor \mathbb{R} \rfloor \mid \lceil \mathbb{R} \rceil \mid \sqrt{\mathbb{R}} \mid \mathbb{R}! \mid \Gamma(\mathbb{R})\end{aligned}$$

[22] The gamma function $\Gamma(x)$ is an extension of the factorial function that accepts real and complex-valued inputs. For positive integers x it produces $(x-1)!$.

We round evaluated expressions to the nearest integer. An expression that is rounded up can be contained inside a ceiling operation, and an expression that is rounded down can be contained inside a floor operation, so all such expressions are valid.

Exercise 23.9. Consider the probabilistic grammar

$$\begin{aligned}\mathbb{R} &\mapsto \mathbb{R} + \mathbb{R} \mid \mathbb{R} \times \mathbb{R} \mid \mathbb{F} \mid \mathbb{I} & w_{\mathbb{R}} &= [1, 1, 5, 5] \\ \mathbb{F} &\mapsto 1.5 \mid \infty & w_{\mathbb{F}} &= [4, 3] \\ \mathbb{I} &\mapsto 1 \mid 2 \mid 3 & w_{\mathbb{I}} &= [1, 1, 1]\end{aligned}$$

What is the generation probability of the expression $1.5 + 2$?

Solution: The expression is obtained by applying the production rules:

$$\begin{aligned}\mathbb{R} &\mapsto \mathbb{R} + \mathbb{R} & P &= 1/12 \\ \mathbb{R} &\mapsto \mathbb{F} & P &= 5/12 \\ \mathbb{F} &\mapsto 1.5 & P &= 4/7 \\ \mathbb{R} &\mapsto \mathbb{I} & P &= 5/12 \\ \mathbb{I} &\mapsto 2 & P &= 1/3\end{aligned}$$

which has a probability of

$$\frac{1}{12}\frac{5}{12}\frac{4}{7}\frac{5}{12}\frac{1}{3} = \frac{25}{9072} \approx 0.00276$$

Exercise 23.10. What is the probabilistic grammar from the previous question after clearing the counts and applying a learning update on $1.5 + 2$?

Solution: The learning update clears all counts and then increments each production rule each time it is applied. The five applied rules are each incremented once, resulting in:

$$\begin{aligned}\mathbb{R} &\mapsto \mathbb{R} + \mathbb{R} \mid \mathbb{R} \times \mathbb{R} \mid \mathbb{F} \mid \mathbb{I} & w_{\mathbb{R}} &= [1, 0, 1, 1] \\ \mathbb{F} &\mapsto 1.5 \mid \infty & w_{\mathbb{F}} &= [1, 0] \\ \mathbb{I} &\mapsto 1 \mid 2 \mid 3 & w_{\mathbb{I}} &= [0, 1, 0]\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

24 Multidisciplinary Optimization

Multidisciplinary design optimization (MDO) involves solving optimization problems spanning across disciplines. Many real-world problems involve complicated interactions between several disciplines, and optimizing disciplines individually may not lead to an optimal solution. This chapter discusses a variety of techniques for taking advantage of the structure of MDO problems to reduce the effort required for finding good designs.[1]

[1] An extensive survey is provided by J. R. R. A. Martins and A. B. Lambe, "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, vol. 51, no. 9, pp. 2049–2075, 2013. Further discussion can be found in J. Sobieszczanski-Sobieski, A. Morris, and M. van Tooren, *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*. Wiley, 2015. See also: N. M. Alexandrov and M. Y. Hussaini, eds., *Multidisciplinary Design Optimization: State of the Art*. SIAM, 1997. J. R. R. A. Martins and A. Ning, *Engineering Design Optimization*. Cambridge University Press, 2022.

24.1 Disciplinary Analyses

There are many different *disciplinary analyses* that might factor into a design. For example, the design of a rocket might involve analysis from disciplines such as structures, aerodynamics, and controls. The different disciplines have their own analytical tools, such as finite element analysis. Often these disciplinary analyses tend to be quite sophisticated and computationally expensive. In addition, disciplinary analyses are often tightly coupled with each other, where one discipline may require the output of another's disciplinary analysis. Resolving these interdependencies can be nontrivial.

In an MDO setting, we still have a set of design variables as before, but we also keep track of the outputs, or *response variables*, of each disciplinary analysis.[2] We write the response variables of the ith disciplinary analysis as $\mathbf{y}^{(i)}$. In general, the ith disciplinary analysis F_i can depend on the design variables or the response variables from any other discipline:

[2] A disciplinary analysis can provide inputs for other disciplines, the objective function, or the constraints. In addition, it can also provide gradient information for the optimizer.

$$\mathbf{y}^{(i)} \leftarrow F_i\left(\mathbf{x}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(i-1)}, \mathbf{y}^{(i+1)}, \ldots, \mathbf{y}^{(m)}\right) \tag{24.1}$$

where m is the total number of disciplines. The inputs to a computational fluid dynamics analysis for an aircraft may include the deflections of the wing, which

come from a structural analysis that requires the forces from computational fluid dynamics. An important part of formulating MDO problems is taking into account such dependencies between analyses.

In order to make reasoning about disciplinary analyses easier, we introduce the concept of an *assignment*. An assignment $\mathcal{A}$ is a set of variable names and their corresponding values relevant to a multidisciplinary design optimization problem. To access a variable v, we write $\mathcal{A}[v]$.

A disciplinary analysis is a function that takes an assignment and uses the design point and response variables from other analyses to overwrite the response variable for its discipline:

$$\mathcal{A}' \leftarrow F_i(\mathcal{A}) \tag{24.2}$$

where $F_i(\mathcal{A})$ updates $\mathbf{y}^{(i)}$ in $\mathcal{A}$ to produce $\mathcal{A}'$.

Assignments can be represented in code using dictionaries.[3] Each variable is assigned a name of type `String`. Variables are not restricted to floating-point numbers but can include other objects, such as vectors. Example 24.1 shows an implementation using a dictionary.

[3] A *dictionary*, also called an *associative array*, is a common data structure that allows indexing by keys rather than integers. See appendix A.1.8.

Consider an optimization with one design variable x and two disciplines. Suppose the first disciplinary analysis F_1 computes a response variable $y^{(1)} = f_1(x, y^{(2)})$ and the second disciplinary analysis F_2 computes a response variable $y^{(2)} = f_2(x, y^{(1)})$.

This problem can be implemented as:

```
function F1(A)
    A["y1"] = f1(A["x"], A["y2"])
    return A
end
function F2(A)
    A["y2"] = f2(A["x"], A["y1"])
    return A
end
```

The assignment may be initialized with guesses for $y^{(1)}$ and $y^{(2)}$, and a known input for x. For example:

```
A = Dict("x"⇒1, "y1"⇒2, "y2"⇒3)
```

Example 24.1. Basic code syntax for the assignment-based representation of multidisciplinary design optimization problems.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

24.2 *Interdisciplinary Compatibility*

Evaluating the objective function value and feasibility of a design point $\mathbf{x}$ requires obtaining values for the response variables that satisfy *interdisciplinary compatibility*, which means that the response variables must be consistent with the disciplinary analyses. Interdisciplinary compatibility holds for a particular assignment if the assignment is unchanged under all disciplinary analyses:

$$F_i(\mathcal{A}) = \mathcal{A} \text{ for } i \text{ in } 1:m \tag{24.3}$$

Running any analysis will produce the same values. Finding an assignment that satisfies interdisciplinary compatibility is called a *multidisciplinary analysis*.

System optimization for a single discipline requires an optimizer to select design variables and query the disciplinary analysis in order to evaluate the constraints and the objective function, as shown in figure 24.1. Single-discipline optimization does not require that we consider disciplinary coupling.

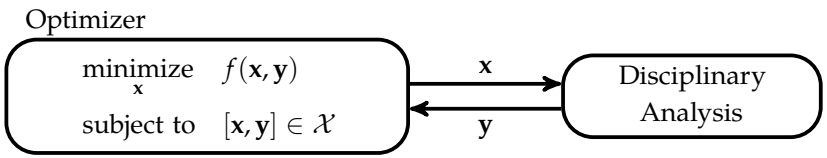


Figure 24.1. Optimization diagram for a single discipline. Gradients may or may not be computed.

System optimization for multiple disciplines can introduce dependencies, in which case coupling becomes an issue. A diagram for two coupled disciplines is given in figure 24.2. Applying conventional optimization to this problem is less straightforward because interdisciplinary compatibility must be established.

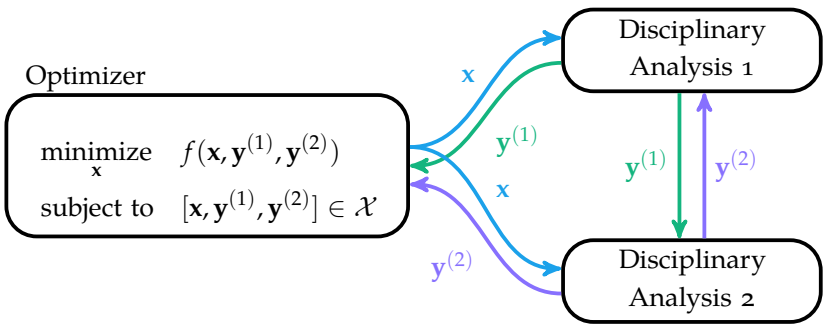
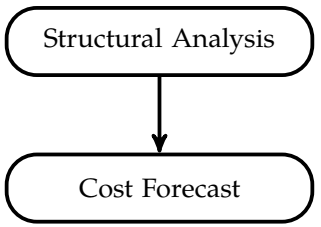


Figure 24.2. Optimization diagram for a two-discipline analysis with interdisciplinary coupling.

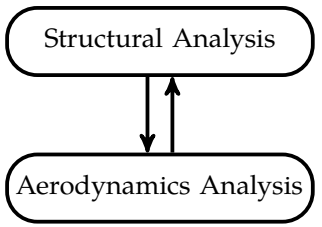
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

If a multidisciplinary analysis does not have a *dependency cycle*,[4] then solving for interdisciplinary compatibility is straightforward. We say discipline i depends on discipline j if i requires any of j's outputs. This dependency relation can be used to form a *dependency graph*, where each node corresponds to a discipline and an edge $j \to i$ is included if discipline i depends on j. Figure 24.3 shows examples of dependency graphs involving two disciplines with and without cycles.

[4] A dependency cycle arises when disciplines depend on each other.



An acyclic dependency graph. An evaluation ordering can be specified such that the required inputs for each discipline are available from previously evaluated disciplines.



A cyclic dependency graph. The structural analysis depends on the aerodynamics analysis and vice versa.

Figure 24.3. Cyclic and acyclic dependency graphs.

If the dependency graph has no cycles, then there always exists an order of evaluation that, if followed, ensures that the necessary disciplinary analyses are evaluated before the disciplinary analyses that depend on them. Such an ordering is called a *topological ordering* and can be found using a topological sorting method such as *Kahn's algorithm*.[5] The reordering of analyses is illustrated in figure 24.4.

If the dependency graph has cycles, then no topological ordering exists. To address cycles, we can use the *Gauss-Seidel method* (algorithm 24.1), which attempts to resolve the multidisciplinary analysis by iterating until convergence.[6] The Gauss-Seidel algorithm is sensitive to the ordering of the disciplines as illustrated by example 24.2. A poor ordering can prevent or slow convergence. The best orderings are those with minimal feedback connections.[7]

It can be advantageous to merge disciplines into a new disciplinary analysis—to group conceptually related analyses, simultaneously evaluate tightly coupled analyses, or more efficiently apply some of the architectures discussed in this chapter. Disciplinary analyses can be merged to form a new analysis whose response variables consist of the response variables of the merged disciplines. The form of the new analysis depends on the disciplinary interdependencies. If

[5] A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[6] The Gauss-Seidel algorithm can also be written to execute analyses in parallel.

[7] In some cases, disciplines can be separated into different clusters that are independent of each other. Each connected cluster can be solved using its own, smaller multidisciplinary analysis.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

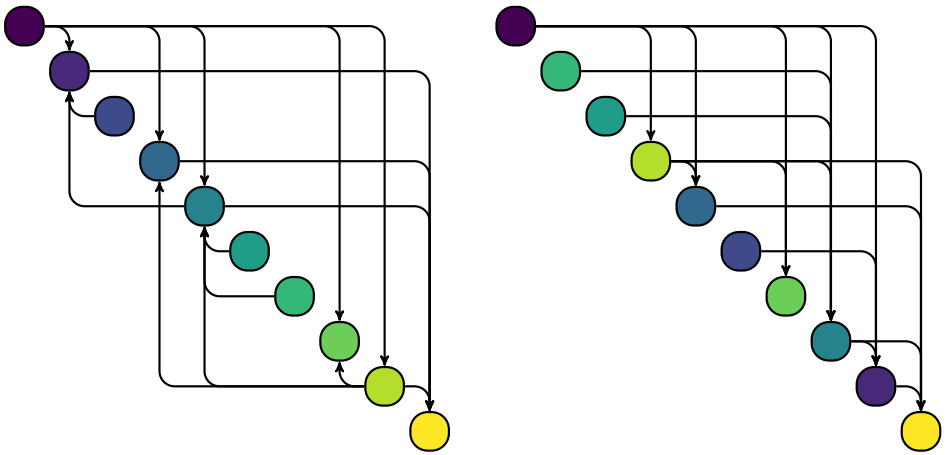


Figure 24.4. A topological sort can be used to reorder the disciplinary analyses to remove feedback connections.

```
function gauss_seidel!(Fs, A; k_max=100, ϵ=1e-4)
    k, converged = 0, false
    while !converged && k ≤ k_max
        k += 1
        A_old = deepcopy(A)
        for F in Fs
            F(A)
        end
        converged = all(isapprox(A[v], A_old[v], rtol=ϵ)
                        for v in keys(A))
    end
    return (A, converged)
end
```

Algorithm 24.1. The Gauss-Seidel algorithm for conducting a multidisciplinary analysis. Here, `Fs` is a vector of disciplinary analysis functions that take and modify an assignment, `A`. There are two optional arguments: the maximum number of iterations `k_max` and the relative error tolerance `ϵ`. The method returns the modified assignment and whether it converged.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 24.2. An example that illustrates the importance of choosing an appropriate ordering when running a multidisciplinary analysis.

Consider a multidisciplinary design optimization problem with one design variable x and three disciplines, each with one response variable:

$$\begin{aligned} y^{(1)} &\leftarrow F_1(x, y^{(2)}, y^{(3)}) = y^{(2)} - x \\ y^{(2)} &\leftarrow F_2(x, y^{(1)}, y^{(3)}) = \sin(y^{(1)} + y^{(3)}) \\ y^{(3)} &\leftarrow F_3(x, y^{(1)}, y^{(2)}) = \cos(x + y^{(1)} + y^{(2)}) \end{aligned}$$

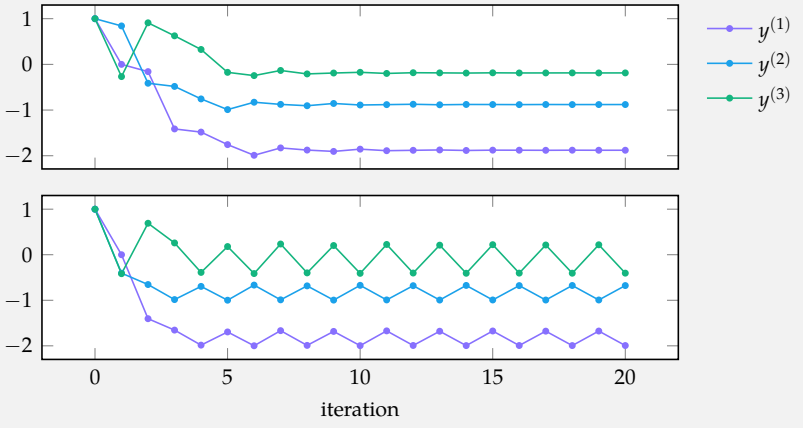
The disciplinary analyses can be implemented as:

```
function F1(A)
    A["y1"] = A["y2"] - A["x"]
    return A
end
function F2(A)
    A["y2"] = sin(A["y1"] + A["y3"])
    return A
end
function F3(A)
    A["y3"] = cos(A["x"] + A["y2"] + A["y1"])
    return A
end
```

Consider running a multidisciplinary analysis for $x = 1$, having initialized our assignment with all 1's:

```
A = Dict("x"⇒1, "y1"⇒1, "y2"⇒1, "y3"⇒1)
```

Running the Gauss-Seidel algorithm with the ordering F_1, F_2, F_3 converges, but running with F_1, F_3, F_2 does not.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

the merged disciplines are acyclic, then an ordering exists in which the analyses can be serially executed. If the merged disciplines are cyclic, then the new analysis must internally run a multidisciplinary analysis to achieve compatibility.

24.3 Architectures

Multidisciplinary design optimization problems can be written:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\mathcal{A}) \\ \text{subject to} \quad & \mathcal{A} \in \mathcal{X} \\ & F_i(\mathcal{A}) = \mathcal{A} \text{ for each discipline } i \in 1:m \end{aligned} \tag{24.4}$$

where the objective function f and feasible set $\mathcal{X}$ depend on both the design and response variables. The design variables in the assignment are specified by the optimizer. The condition $F_i(\mathcal{A}) = \mathcal{A}$ ensures that the ith discipline is consistent with the values in $\mathcal{A}$. This last condition enforces interdisciplinary compatibility.

There are several challenges associated with optimizing multidisciplinary problems. The interdependence of disciplinary analyses causes the ordering of analyses to matter and often makes parallelization difficult or impossible. There is a trade-off between an optimizer that directly controls all variables and incorporating suboptimizers[8] that leverage discipline-specific expertise to optimize values locally. In addition, there is a trade-off between the expense of running disciplinary analyses and the expense of globally optimizing too many variables. Finally, every architecture must enforce interdisciplinary compatibility in the final solution.

[8] A *suboptimizer* is an optimization routine called within another optimization routine.

The remainder of this chapter discusses a variety of different optimization architectures for addressing these challenges. These architectures are demonstrated using the hypothetical ride-sharing problem introduced in example 24.3.

24.4 Multidisciplinary Design Feasible

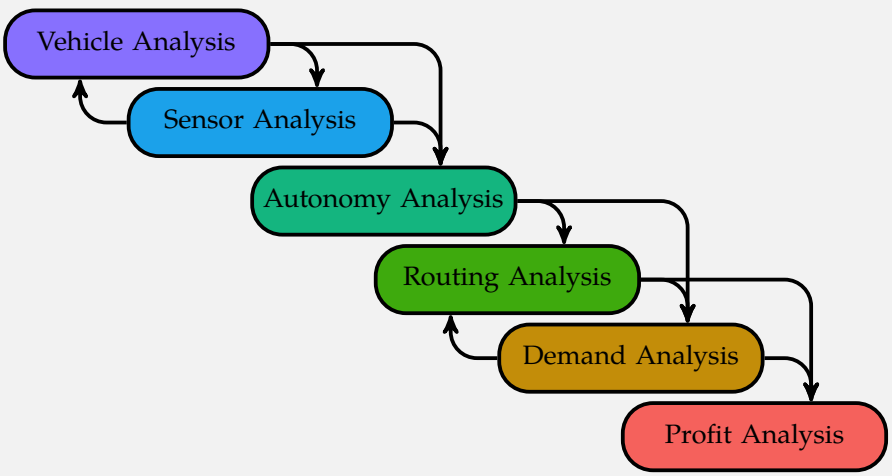
The *multidisciplinary design feasible* architecture structures the MDO problem such that standard optimization algorithms can be directly applied to optimize the design variables. A multidisciplinary design analysis is run for any given design point to obtain compatible response values.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 24.3. A ride-sharing problem used throughout this chapter to demonstrate optimization architectures.

Consider a ride-sharing company developing a self-driving fleet. This hypothetical company is simultaneously designing the vehicle, its sensor package, a routing strategy, and a pricing scheme. These portions of the design are referred to as $\mathbf{v}$, $\mathbf{s}$, $\mathbf{r}$, and $\mathbf{p}$, respectively, each of which contains numerous design variables. The vehicle, for instance, may include parameters governing the structural geometry, engine and drive train, battery capacity, and passenger capacity.

The objective of the ride-sharing company is to maximize profit. The profit depends on a large-scale simulation of the routing algorithm and passenger demand, which, in turn, depends on response variables from an autonomy analysis of the vehicle and its sensor package. Several analyses extract additional information. The performance of the routing algorithm depends on the demand generated by the pricing scheme, and the demand generated by the pricing scheme depends on performance of the routing algorithm. The vehicle range and fuel efficiency depend on the weight, drag, and power consumption of the sensor package. The sensor package requires vehicle geometry and performance information to meet the necessary safety requirements. A dependency diagram is presented below.



© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

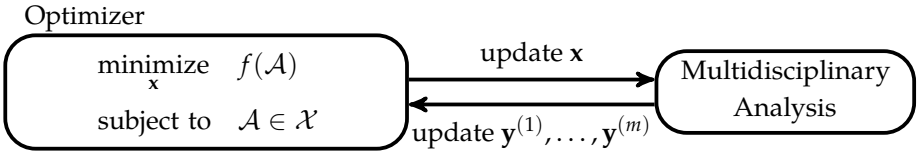


Figure 24.5. The multidisciplinary design feasible architecture. The optimizer chooses design points $\mathbf{x}$, and the multidisciplinary analysis computes a consistent assignment $\mathcal{A}$. The structure is similar to that of single-discipline optimization.

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f\left(\mathbf{x}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}\right) \\ \text{subject to} \quad & \left[\mathbf{x}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}\right] \in \mathcal{X} \end{aligned} \quad \longrightarrow \quad \begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\text{MDA}(\mathbf{x})) \\ \text{subject to} \quad & \text{MDA}(\mathbf{x}) \in \mathcal{X} \end{aligned}$$

Figure 24.6. Transforming an MDO problem into a typical optimization problem using multidisciplinary design analyses, where $\text{MDA}(\mathbf{x})$ returns a multidisciplinary compatible assignment.

An architecture diagram is given in figure 24.5. It consists of two blocks, the optimizer and the multidisciplinary analysis. The optimizer is the method used for selecting design points with the goal of minimizing the objective function. The optimizer calls the multidisciplinary analysis block by passing it a design point $\mathbf{x}$ and receives a compatible assignment $\mathcal{A}$. If interdisciplinary compatibility is not possible, the multidisciplinary analysis block informs the optimizer and such design points are treated as infeasible. Figure 24.6 shows how an MDO problem can be transformed into a typical optimization problem using multidisciplinary design analyses.

The primary advantages of the multidisciplinary design feasible architecture are its conceptual simplicity and that it is guaranteed to maintain interdisciplinary compatibility at each step in the optimization. Its name reflects the fact that multidisciplinary design analyses are run at every design evaluation, ensuring that the system-level optimizer only considers feasible designs.

The primary disadvantage is that multidisciplinary design analyses are expensive to run, typically requiring several iterations over all analyses. Iterative Gauss-Seidel methods may be slow to converge or may not converge at all, depending on the initialization of the response variables and the ordering of the disciplinary analyses.

Lumping the analyses together makes it necessary for all local variables—typically only relevant to a particular discipline—to be considered by the analysis as a whole. Many practical problems have a very large number of local design variables, such as mesh control points in aerodynamics, element dimensions in structures, component placements in electrical engineering, and neural network

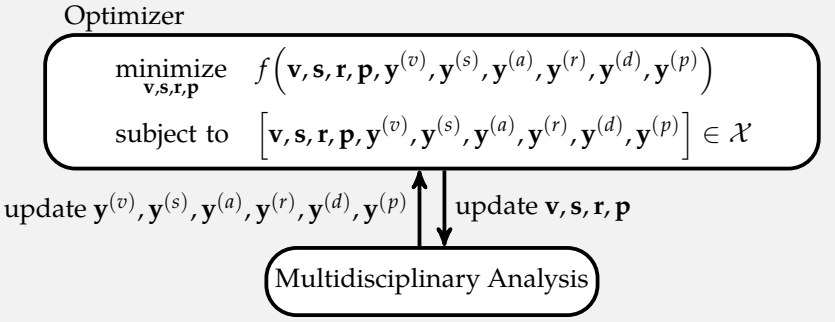
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

weights in machine learning. Multidisciplinary design feasible optimization requires that the system optimizer specify all of these values across all disciplines while satisfying all constraints.

The multidisciplinary design feasible architecture is applied to the ride-sharing problem in example 24.4.

Example 24.4. The multidisciplinary design feasible architecture applied to the ride-sharing problem. A multidisciplinary analysis over all response variables must be completed for every candidate design point. This tends to be very computationally intensive.

The multidisciplinary design feasible architecture can be applied to the ride-sharing problem. The architectural diagram is shown below.



24.5 Sequential Optimization

The *sequential optimization* architecture (figure 24.7) is an architecture that can leverage discipline-specific tools and experience to optimize subproblems but can lead to suboptimal solutions. This architecture is included to demonstrate the limitations of a naive approach and to serve as a baseline against which other architectures can be compared.

A *subproblem* is an optimization procedure conducted at every iteration of an overarching optimization process. Sometimes design variables can be removed from the outer optimization procedure, the *system-level optimizer*, and can be more efficiently optimized in subproblems.

The design variables for the ith discipline can be partitioned according to $\mathbf{x}^{(i)} = [\mathbf{x}_g^{(i)}, \mathbf{x}_\ell^{(i)}]$, where $\mathbf{x}_g^{(i)}$ are *global design variables* shared with other disciplines and $\mathbf{x}_\ell^{(i)}$ are *local design variables* used only by the associated disciplinary subproblem.[9] The response variables can be similarly partitioned into the global response variables $\mathbf{y}_g^{(i)}$ and the local response variables $\mathbf{y}_\ell^{(i)}$. Disciplinary autonomy is

[9] The vehicle subproblem in the ride-sharing problem may include global design variables, such as the vehicle capacity and range, that affect other disciplines but may also include local design variables, such as the seating configuration, that do not impact other disciplines.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

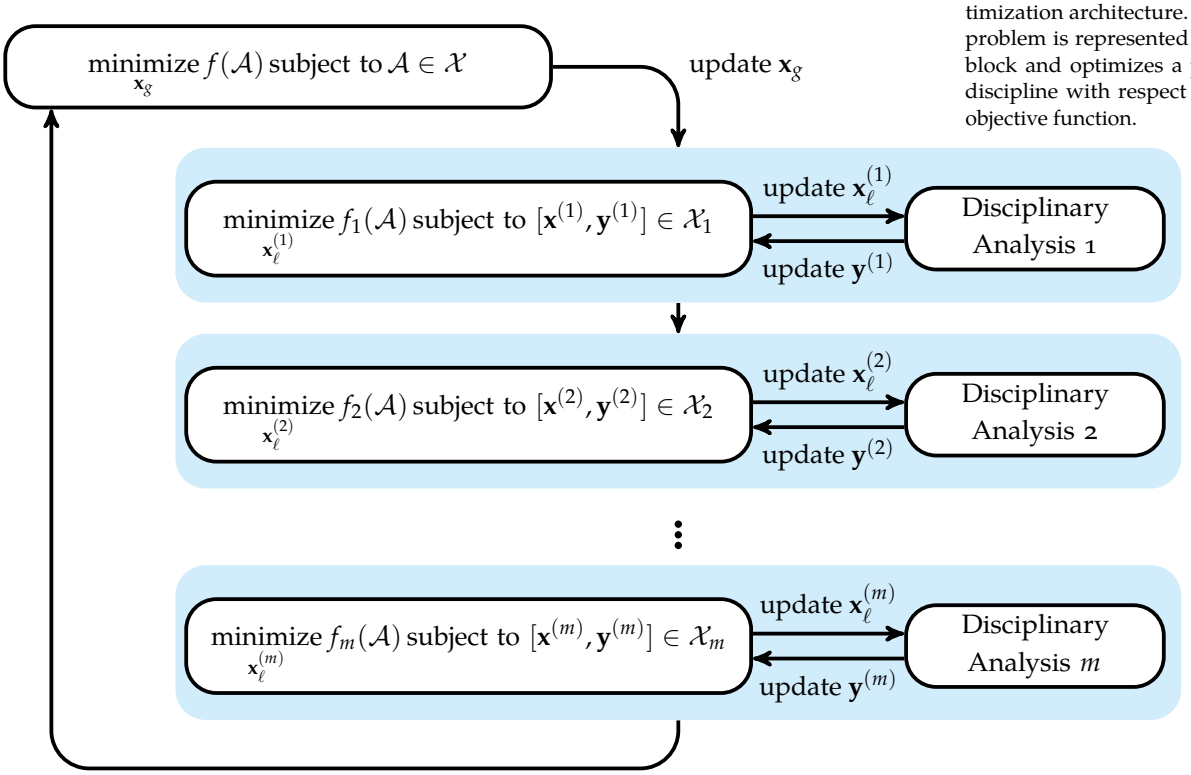


Figure 24.7. The sequential optimization architecture. Each subproblem is represented by a blue block and optimizes a particular discipline with respect to a local objective function.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

achieved by optimizing the local variables in their own disciplinary optimizers. A local objective function f_i must be chosen such that optimizing it also benefits the global objective. A top-level optimizer is responsible for optimizing the global design variables $\mathbf{x}_g$ with respect to the original objective function. An instantiation of $\mathbf{x}_g$ is evaluated through sequential optimizations; each subproblem is optimized one after the other, passing its results to the next subproblem until all have been evaluated.

Sequential optimization takes advantage of the locality of disciplines: that many variables are unique to a particular discipline and do not need to be shared across discipline boundaries. Sequential optimization harnesses each discipline's proficiency at solving its discipline-specific problem. The subproblem optimizers have complete control over their discipline-specific design variables to meet local design objectives and constraints.

Except in special cases, sequential optimization does not lead to an optimal solution of the original problem for the same reason that Gauss-Seidel is not guaranteed to converge. The solution is sensitive to the local objective functions, and finding suitable local objective functions is often a challenge. Sequential optimization does not support parallel execution, and interdisciplinary compatibility is enforced through iteration and does not always converge.

Example 24.5 applies sequential optimization to the ride-sharing problem.

24.6 Individual Discipline Feasible

The *individual discipline feasible* (IDF) architecture removes the need to run expensive multidisciplinary design analyses and allows disciplinary analyses to be executed in parallel. It loses the guarantee that interdisciplinary compatibility is maintained throughout its execution, with eventual agreement enforced through equality constraints in the optimizer. Compatibility is not enforced in multidisciplinary analyses but rather by the optimizer itself.

IDF introduces *coupling variables* to the design space. For each discipline, an additional vector $\mathbf{c}^{(i)}$ is added to the optimization problem to act as aliases for the response variables $\mathbf{y}^{(i)}$. The response variables are unknown until they are computed by their respective domain analyses; inclusion of the coupling variables allows the optimizer to provide these estimates to multiple disciplines simultaneously when running analyses in parallel. Equality between the cou-

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 24.5. Sequential optimization for ride-sharing.

The sequential optimization architecture can optimize some variables locally. Figure 24.8 shows the result of applying sequential optimization to the ride-sharing problem.

The design variables for the vehicle, sensor system, routing algorithm, and pricing scheme are split into local discipline-specific variables and top-level global variables. For example, the vehicle subproblem can optimize local vehicle parameters $\mathbf{v}_\ell$ such as drive train components, whereas parameters like vehicle capacity that are used by other analyses are controlled globally in $\mathbf{v}_g$.

The tight coupling between the vehicle and sensor systems is poorly handled by the sequential optimization architecture. While changes made by the vehicle subproblem are immediately addressed by the sensor subproblem, the effect of the sensor subproblem on the vehicle subproblem is not addressed until the next iteration.

Not all analyses require their own subproblems. The profit analysis is assumed not to have any local design variables and can thus be executed without needing a subproblem block.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

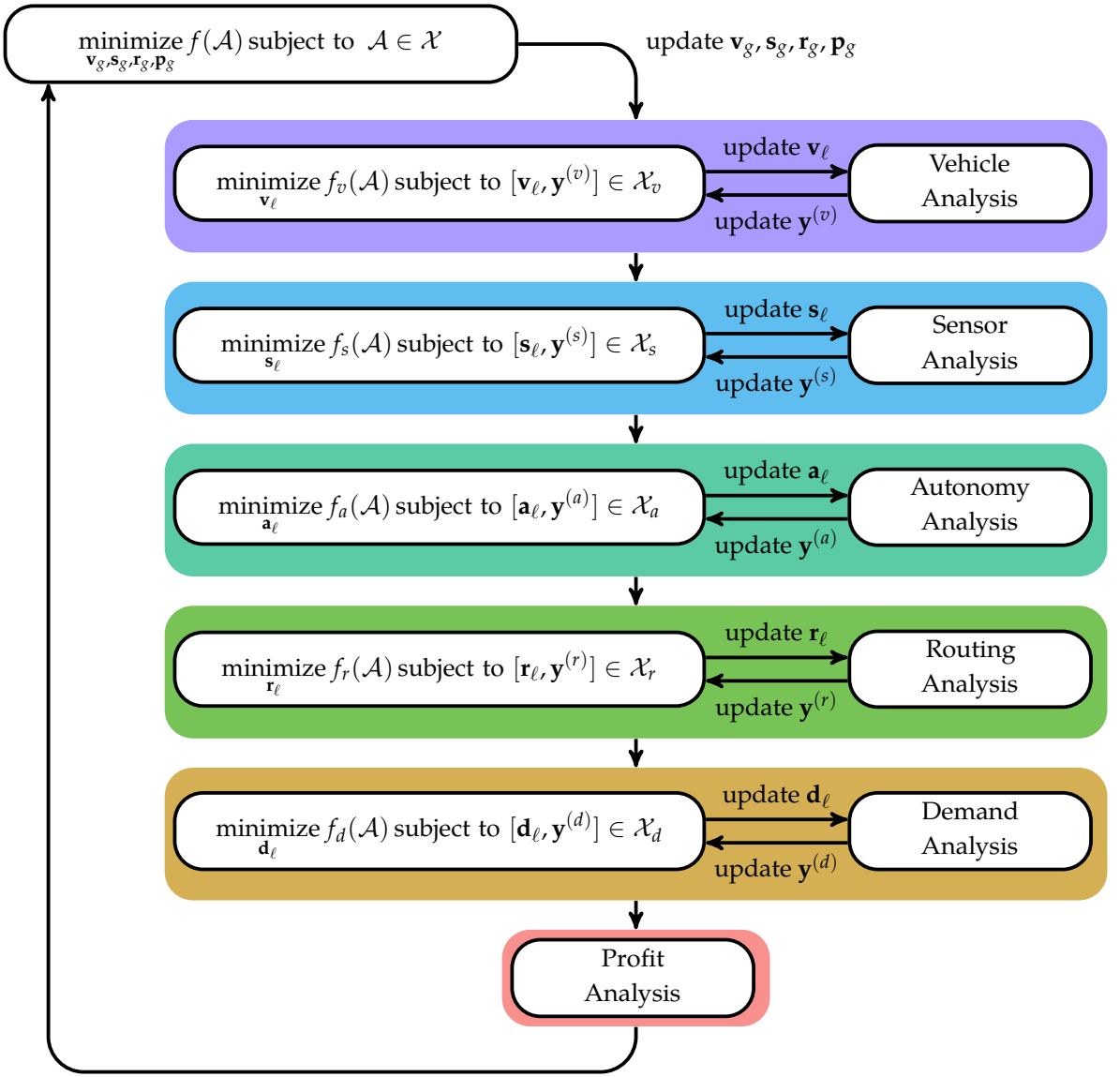


Figure 24.8. The sequential optimization architecture applied to the ride-sharing problem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

pling and response variables is typically reached through iteration. Equality is an optimization constraint, $\mathbf{c}^{(i)} = \mathbf{y}^{(i)}$, for each discipline.

Figure 24.9 shows the general IDF architecture. The system-level optimizer operates on the coupling variables and uses these to populate an assignment that is copied to the disciplinary analysis in each iteration:

$$\mathcal{A}[\mathbf{x}, \mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(m)}] \leftarrow [\mathbf{x}, \mathbf{c}^{(1)}, \ldots, \mathbf{c}^{(m)}] \tag{24.5}$$

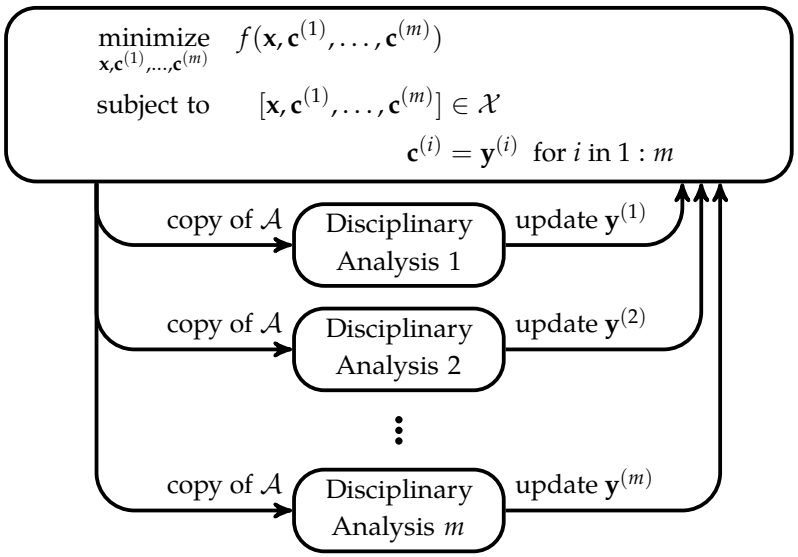


Figure 24.9. The individual discipline feasible architecture allows disciplinary analyses to be run in parallel. Because disciplinary analyses can mutate their inputs, copies of the system-level optimizer's assignment are passed to each disciplinary analysis.

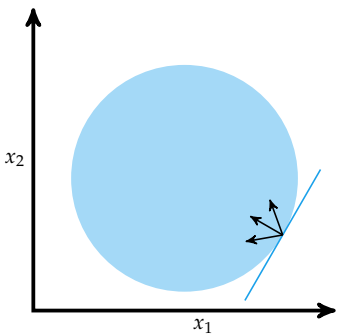


Figure 24.10. The search direction for a point on a constraint boundary must lead into the feasible set.

Despite the added freedom to execute analyses in parallel, IDF suffers from the shortcoming that it cannot leverage domain-specific optimization procedures in the same way as sequential optimization as optimization is top-level only. Furthermore, the optimizer must satisfy additional equality constraints and has more variables to optimize. IDF can have difficulties with gradient-based optimization since the chosen search direction must take constraints into account as shown in figure 24.10. Changes in the design variables must not cause the coupling variables to become infeasible with respect to the disciplinary analyses. Evaluating the gradients of the objective and constraint function is very costly when the disciplinary analyses are expensive.

The individual discipline feasible architecture is applied to the ride-sharing problem in figure 24.11.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

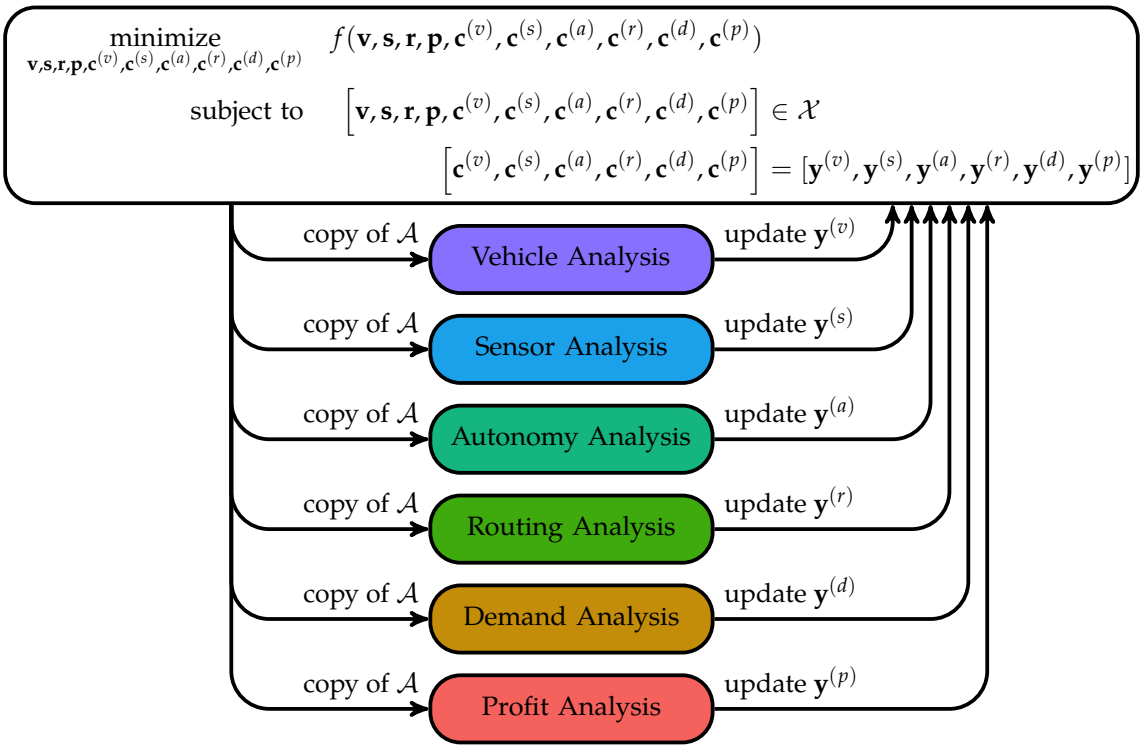


Figure 24.11. The individual discipline feasible architecture applied to the ride-sharing problem. The individual discipline feasible architecture allows for parallel execution of analyses, but the system-level optimizer must optimize a large number of variables.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

24.7 Collaborative Optimization

The *collaborative optimization* architecture (figure 24.12) breaks a problem into disciplinary subproblems that have full control over their local design variables and discipline-specific constraints. Subproblems can be solved using discipline-specific tools and can be optimized in parallel.

The ith subproblem has the form:

$$\begin{aligned} \underset{\mathbf{x}^{(i)}}{\text{minimize}} \quad & f_i(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \\ \text{subject to} \quad & [\mathbf{x}^{(i)}, \mathbf{y}^{(i)}] \in \mathcal{X}_i \end{aligned} \tag{24.6}$$

with $\mathbf{x}^{(i)}$ containing a subset of the design variables $\mathbf{x}$ and response variables $\mathbf{y}^{(i)}$. The constraint ensures that the solution satisfies discipline-specific constraints.

Interdisciplinary compatibility requires that the global variables $\mathbf{x}_g^{(i)}$ and $\mathbf{y}_g^{(i)}$ agree between all disciplines. We define a set of coupling variables $\mathcal{A}_g$ that includes variables corresponding to all design and response variables that are global in at least one subproblem. Agreement is enforced by constraining each $\mathbf{x}_g^{(i)}$ and $\mathbf{y}_g^{(i)}$ to match its corresponding coupling variables:

$$\mathbf{x}_g^{(i)} = \mathcal{A}_g[\mathbf{x}_g^{(i)}] \qquad \text{and} \qquad \mathbf{y}_g^{(i)} = \mathcal{A}_g[\mathbf{y}_g^{(i)}] \tag{24.7}$$

where $\mathcal{A}_g[\mathbf{x}_g^{(i)}]$ and $\mathcal{A}_g[\mathbf{y}_g^{(i)}]$ are the coupling variables corresponding to the global design and response variables in the ith discipline. This constraint is enforced using the subproblem objective function:

$$f_i = \left\| \mathbf{x}_g^{(i)} - \mathcal{A}_g[\mathbf{x}_g^{(i)}] \right\|_2^2 + \left\| \mathbf{y}_g^{(i)} - \mathcal{A}_g[\mathbf{y}_g^{(i)}] \right\|_2^2 \tag{24.8}$$

Each subproblem thus seeks feasible solutions that minimally deviate from the coupling variables.

The subproblems are managed by a system-level optimizer that is responsible for optimizing the coupling variables $\mathcal{A}_g$ to minimize the objective function. Evaluating an instance of the coupling variables requires running each disciplinary subproblem, typically in parallel.

Disciplinary subproblems may deviate from the coupling variables during the optimization process. This discrepancy occurs when two or more disciplines disagree on a variable or when subproblem constraints prevent matching the target values set by the system-level optimizer. The top-level constraint that $f_i = 0$ for each discipline ensures that coupling is eventually attained.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

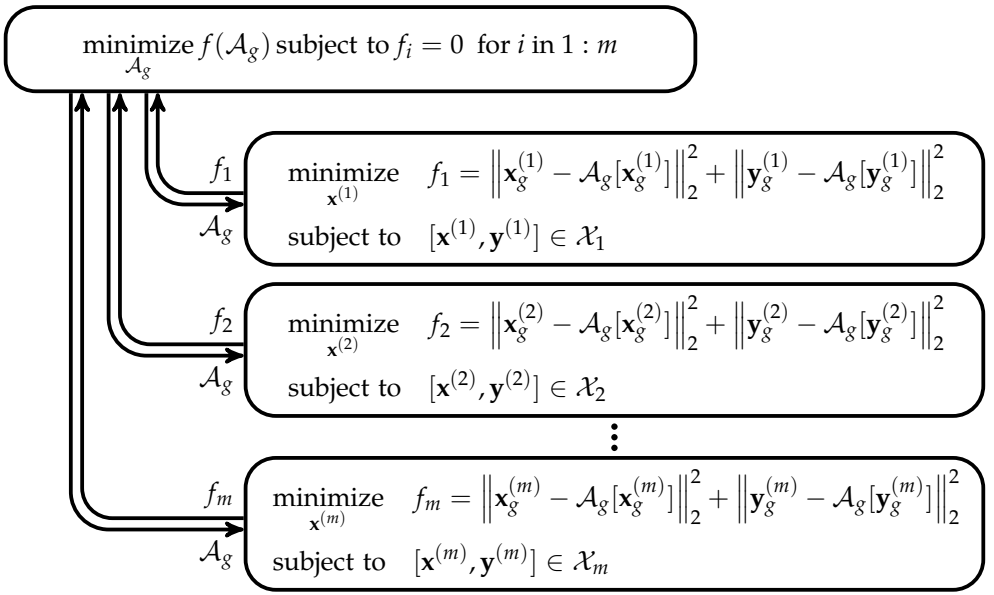


Figure 24.12. Design architecture for collaborative optimization.

The primary advantages of collaborative optimization stem from its ability to isolate some design variables into disciplinary subproblems. Collaborative optimization is readily applicable to real-world multidisciplinary problem-solving, as each discipline is typically well segregated, and thus largely unaffected by small decisions made in other disciplines. The decentralized formulation allows traditional discipline optimization methods to be applied, allowing problem designers to leverage existing tools and methodologies.

Collaborative optimization requires optimizing over the coupling variables, which includes both design and response variables. Collaborative optimization does not perform well in problems with high coupling because the additional coupling variables can outweigh the benefits of local optimization.

Collaborative optimization is a *distributed architecture* that decomposes a single optimization problem into a set of smaller optimization problems that have the same solution when their solutions are combined. Distributed architectures have the advantage of reduced solving times, as subproblems can be optimized in parallel.

Collaborative optimization is applied to the ride-sharing problem in example 24.6.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example 24.6. Applying collaborative optimization to the ride-sharing problem.

The collaborative optimization architecture can be applied to the ride-sharing problem by producing six different disciplinary subproblems. Unfortunately, having six different subproblems requires any variables shared across disciplines to be optimized at the global level.

Figure 24.13 shows two disciplinary subproblems obtained by grouping the vehicle, sensor, and autonomy disciplines into a transport subproblem and the routing, demand, and profit disciplines into a network subproblem. The disciplines grouped into each subproblem are tightly coupled. Having only two subproblems significantly reduces the number of global variables considered by the system-level optimizer because presumably very few design variables are directly used by both the transport and network subproblems.

The subproblems are each multidisciplinary optimization problems, themselves amenable to optimization using the techniques covered in this chapter. We can, for example, use sequential optimization within the transport subproblem. We can also add another instance of collaborative optimization within the network subproblem.

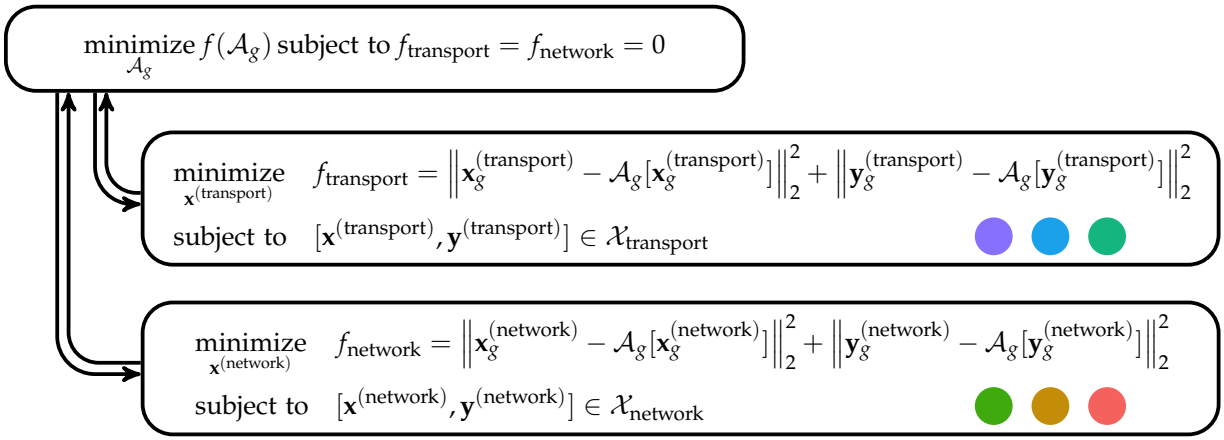


Figure 24.13. The collaborative optimization architecture applied to the ride-sharing problem. The colored circles correspond to the disciplinary analyses contained within each subproblem.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

24.8 Simultaneous Analysis and Design

The *simultaneous analysis and design* (SAND) architecture avoids the central challenge of coordinating between multiple disciplinary analyses by having the optimizer conduct the analyses. Instead of running an analysis $F_i(\mathcal{A})$ to obtain a residual, SAND optimizes both the design and response variables subject to a constraint $F_i(\mathcal{A}) = \mathcal{A}$. The optimizer is responsible for simultaneously optimizing the design variables and finding the corresponding response variables.

Any disciplinary analysis can be transformed into *residual form*. The residual $r_i(\mathcal{A})$ is used to indicate whether an assignment $\mathcal{A}$ is compatible with the ith discipline. If $F_i(\mathcal{A}) = \mathcal{A}$, then $r_i(\mathcal{A}) = 0$; otherwise, $r_i(\mathcal{A}) \neq 0$. We can obtain a residual form using the disciplinary analysis:

$$r_i(\mathcal{A}) = \left\| F_i(\mathcal{A}) - \mathcal{A}[\mathbf{y}^{(i)}] \right\| \tag{24.9}$$

though this is typically inefficient, as demonstrated in example 24.7.

Example 24.7. Evaluating a disciplinary analysis in a residual is typically counterproductive. The analysis must typically perform additional work to solve the problem whereas a cleaner residual form can more efficiently verify whether the inputs are compatible.

Consider a disciplinary analysis that solves the equation $\mathbf{A}\mathbf{y} = \mathbf{x}$. The analysis is $F(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$, which requires an expensive matrix inversion. We can construct a residual form using equation (24.9):

$$r_1(\mathbf{x}, \mathbf{y}) = \|F(\mathbf{x}) - \mathbf{y}\| = \left\| \mathbf{A}^{-1}\mathbf{x} - \mathbf{y} \right\|$$

Alternatively, we can use the original constraint to construct a more efficient residual form:

$$r_2(\mathbf{x}, \mathbf{y}) = \|\mathbf{A}\mathbf{y} - \mathbf{x}\|$$

The residual form of a discipline consists of the set of disciplinary equations that are solved by the disciplinary analysis.[10] It is often much easier to evaluate a residual than to run a disciplinary analysis. In SAND, figure 24.14, the analysis effort is the responsibility of the optimizer.

[10] In aerodynamics, these may include the Navier-Stokes equations. In structural engineering, these may include the elasticity equations. In electrical engineering, these may include the differential equations for current flow.

$$\underset{\mathcal{A}}{\text{minimize}}\ f(\mathcal{A}) \text{ subject to } \mathcal{A} \in \mathcal{X},\ r_i(\mathcal{A}) = 0 \text{ for each discipline}$$

Figure 24.14. Simultaneous analysis and design places the entire burden on the optimizer. It uses disciplinary residuals rather than disciplinary analyses.

SAND can explore regions of the design space that are infeasible with respect to the residual equations, as shown in figure 24.15. Exploring infeasible regions

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

can allow us to traverse the design space more easily and find solutions in feasible regions disconnected from the feasible region of the starting design point. SAND suffers from having to simultaneously optimize a very large number of variables for which derivatives and other discipline-specific expertise are not available. Furthermore, SAND gains much of its value from residuals that can be computed more efficiently than can disciplinary analyses. Use of SAND in real-world applications is often limited by the inability to modify existing disciplinary analysis code to produce an efficient residual form.

SAND is applied to the ride-sharing problem in example 24.8.

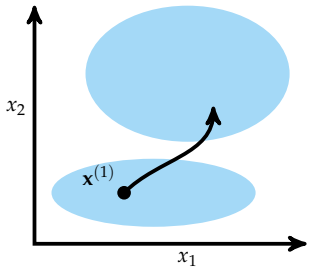


Figure 24.15. SAND can explore regions of the design space that are infeasible and potentially bridge the gap between feasible subsets.

Example 24.8. The simultaneous analysis and design architecture applied to the ride-sharing problem.

Applying the simultaneous analysis and design architecture to the ride-sharing problem requires disciplinary residuals. These can potentially depend on all design and response variables. The architecture requires that the optimizer optimize all of the design variables and all of the response variables.

$$
\begin{aligned}
\underset{\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}}{\text{minimize}} \quad & f\left(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}\right) \\
\text{subject to} \quad & \left[\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}\right] \in \mathcal{X} \\
& r_v(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0 \\
& r_s(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0 \\
& r_a(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0 \\
& r_r(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0 \\
& r_d(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0 \\
& r_p(\mathbf{v},\mathbf{s},\mathbf{r},\mathbf{p},\mathbf{y}^{(v)},\mathbf{y}^{(s)},\mathbf{y}^{(a)},\mathbf{y}^{(r)},\mathbf{y}^{(d)},\mathbf{y}^{(p)}) = 0
\end{aligned}
$$

24.9 Summary

- Multidisciplinary design optimization requires reasoning about multiple disciplines and achieving agreement between coupled variables.
- Disciplinary analyses can often be ordered to minimize dependency cycles.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

- Multidisciplinary design problems can be structured in different architectures that take advantage of problem features to improve the optimization process.
- The multidisciplinary design feasible architecture maintains feasibility and compatibility through the use of slow and potentially nonconvergent multidisciplinary design analyses.
- Sequential optimization allows each discipline to optimize its discipline-specific variables but does not always yield optimal designs.
- The individual discipline feasible architecture allows parallel execution of analyses at the expense of adding coupling variables to the global optimizer.
- Collaborative optimization incorporates suboptimizers that can leverage domain specialization to optimize some variables locally.
- The simultaneous analysis and design architecture replaces design analyses with residuals, allowing the optimizer to find compatible solutions, but cannot directly use disciplinary solution techniques.

24.10 Exercises

Exercise 24.1. Provide an example of a practical engineering problem that is multidisciplinary.

Solution: Maximize the lift-to-drag ratio of an airfoil shape subject to a constraint on the structural stability of the airfoil.

Exercise 24.2. Provide an abstract example of a multidisciplinary problem where the order of the analyses is important.

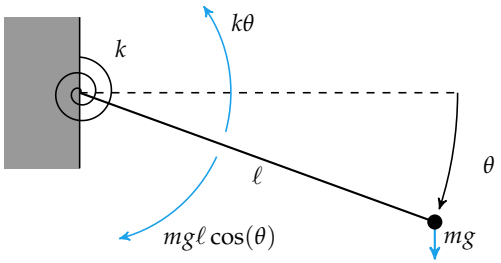
Solution: Consider a problem where the disciplinary dependency graph is a (directed) tree: If the optimization starts from the root and proceeds by following the topological order, then convergence occurs after one traversal of the tree.

Exercise 24.3. What is one advantage of the individual discipline feasible architecture over the multidisciplinary design feasible and sequential optimization architectures?

Solution: It can execute disciplinary analyses in parallel.

Exercise 24.4. Consider applying multidisciplinary design analysis to minimizing the weight of a wing whose deformation and loading are computed by separate disciplines. We will use a simplified version of the problem, representing the wing as a horizontally mounted pendulum supported by a torsional spring.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com



The objective is to minimize the spring stiffness, k, such that the pendulum's displacement does not exceed a target threshold. The pendulum length ℓ, pendulum point mass m, and gravitational constant g are fixed.

We use two simplified analyses in place of more sophisticated analyses used to compute deformations and loadings of aircraft wings. Assuming the pendulum is rigid, the loading moment M is equal to $mg\ell\cos(\theta)$. The torsional spring resists deformation such that the pendulum's angular displacement θ is M/k.

Formulate the spring-pendulum problem under the multidisciplinary design feasible architecture, and then solve it according to that architecture for:

$$m = 1\,\text{kg},\ \ell = 1\,\text{m},\ g = 9.81\,\text{m/s}^2,\ \theta_{\text{max}} = 10^\circ$$

Solution: The spring-pendulum problem under the multidisciplinary design feasible architecture is:

$$\begin{aligned} \underset{k}{\text{minimize}} \quad & f(\text{MDA}(k)) \\ \text{subject to} \quad & k > 0 \\ & \mathcal{A}[\theta] \leq \theta_{\text{max}} \\ & \text{MDA}(k) \ \text{ converged} \end{aligned}$$

where $\text{MDA}(k)$ performs a multidisciplinary analysis on the two disciplinary analyses: a loads analysis that computes $\mathcal{A}[M] = mg\ell\cos(\mathcal{A}[\theta])$ and a displacement analysis that computes $\mathcal{A}[\theta] = \mathcal{A}[M]/\mathcal{A}[k]$. Whether the multidisciplinary design analysis converged is included as an additional response variable in order to enforce convergence.

Solving the optimization problem produces $k \approx 55.353\,\text{N}\cdot\text{m}$.

Exercise 24.5. Formulate the spring-pendulum problem under the individual discipline feasible architecture.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solution: The spring-pendulum problem under the individual discipline feasible architecture is:

$$\begin{aligned}
\underset{k,\theta_c,M_c}{\text{minimize}} \quad & k \\
\text{subject to} \quad & k > 0 \\
& \theta_c = F_{\text{displacement}}(k, M_c) \\
& M_c = F_{\text{loads}}(\theta_c) \\
& \theta_c \leq \theta_{\max}
\end{aligned}$$

where θ_c and M_c are the additional coupling variables under control of the optimizer. The two disciplinary analyses can be executed in parallel.

Exercise 24.6. Formulate the spring-pendulum problem under the collaborative optimization architecture. Present the two disciplinary optimization problems and the system-level optimization problem.

Solution: The two disciplinary optimization problems for the spring-pendulum problem under the collaborative optimization architecture are:

$$\begin{aligned}
\underset{k,M}{\text{minimize}} \quad & f_{\text{displacement}} = (k - \mathcal{A}_g[k])^2 + (M - \mathcal{A}_g[M])^2 + \\
& \qquad (F_{\text{displacement}}(k, M) - \mathcal{A}_g[\theta])^2 \\
\text{subject to} \quad & F_{\text{displacement}}(k, M) \leq \theta_{\max} \\
& k > 0
\end{aligned}$$

and

$$\underset{\theta}{\text{minimize}} \quad f_{\text{loads}} = (\theta - \mathcal{A}_g[\theta])^2 + (F_{\text{loads}}(\theta) - \mathcal{A}_g[M])^2$$

where each subproblem controls its own local copies of the variables it shares with other disciplines, and $\mathcal{A}_g[\cdot]$ is the corresponding coupling variable held by the system-level optimizer. The coupling variables are $\mathcal{A}_g = \{k, \theta, M\}$.

The system-level optimization problem is:

$$\begin{aligned}
\underset{\mathcal{A}_g}{\text{minimize}} \quad & \mathcal{A}_g[k] \\
\text{subject to} \quad & f_{\text{displacement}} = 0 \\
& f_{\text{loads}} = 0
\end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

APPENDICES

A Julia

Julia is a scientific programming language that is free and open source.[1] It is a relatively new language that borrows inspiration from languages like Python, MATLAB, and R. It was selected for use in this book because it is sufficiently high level[2] so that the algorithms can be compactly expressed and readable while also being fast. This book is compatible with Julia version 1.10. This appendix introduces the concepts necessary for understanding the code included in the text, omitting many of the advanced features of the language.

[1] Julia may be obtained from `http://julialang.org`.

[2] In contrast with languages like C++, Julia does not require programmers to worry about memory management and other lower-level details, yet it allows low-level control when needed.

A.1 Types

Julia has a variety of basic types that can represent data given as truth values, numbers, strings, arrays, tuples, and dictionaries. Users can also define their own types. This section explains how to use some of the basic types and how to define new types.

A.1.1 Booleans

The *Boolean* type in Julia, written as `Bool`, includes the values `true` and `false`. We can assign these values to variables. Variable names can be any string of characters, including Unicode, with a few restrictions.

```
α = true
done = false
```

The variable name appears on the left side of the equal sign; the value that variable is to be assigned is on the right side.

We can make assignments in the Julia console. The console, or REPL (for read, eval, print, loop), will return a response to the expression being evaluated. The # symbol indicates that the rest of the line is a comment.

```
julia> x = true
true
julia> y = false; # semicolon suppresses the console output
julia> typeof(x)
Bool
julia> x == y # test for equality
false
```

The standard Boolean operators are supported:

```
julia> !x     # not
false
julia> x && y # and
false
julia> x || y # or
true
```

A.1.2 Numbers

Julia supports integer and floating-point numbers, as shown here:

```
julia> typeof(42)
Int64
julia> typeof(42.0)
Float64
```

Here, `Int64` denotes a 64-bit integer, and `Float64` denotes a 64-bit floating-point value.[3] We can perform the standard mathematical operations:

[3] On 32-bit machines, an integer literal like `42` is interpreted as an `Int32`.

```
julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
8
julia> x / y
2.0
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
julia> x ^ y # exponentiation
16
julia> x % y # remainder from division
0
julia> div(x, y) # truncated division returns an integer
2
```

Note that the result of `x / y` is a `Float64`, even when `x` and `y` are integers. We can also perform these operations at the same time as an assignment. For example, `x += 1` is shorthand for `x = x + 1`.

We can also make comparisons:

```
julia> 3 > 4
false
julia> 3 >= 4
false
julia> 3 ≥ 4   # unicode also works, use \ge[tab] in console
false
julia> 3 < 4
true
julia> 3 <= 4
true
julia> 3 ≤ 4   # unicode also works, use \le[tab] in console
true
julia> 3 == 4
false
julia> 3 < 4 < 5
true
```

A.1.3 Strings

A *string* is an array of characters. Strings are not used very much in this textbook except to report certain errors. An object of type `String` can be constructed using `"` characters. For example:

```
julia> x = "optimal"
"optimal"
julia> typeof(x)
String
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A.1.4 Vectors

A *vector* is a one-dimensional array that stores a sequence of values. We can construct a vector using square brackets, separating elements by commas:

```
julia> x = [];                      # empty vector
julia> x = trues(3);                # Boolean vector containing three trues
julia> x = ones(3);                 # vector of three ones
julia> x = zeros(3);                # vector of three zeros
julia> x = rand(3);                 # vector of three random numbers between 0 and 1
julia> x = [3, 1, 4];               # vector of integers
julia> x = [3.1415, 1.618, 2.7182]; # vector of floats
```

An *array comprehension* can be used to create vectors:

```
julia> [sin(x) for x in 1:5]
5-element Vector{Float64}:
  0.841471
  0.909297
  0.14112
 -0.756802
 -0.958924
```

We can inspect the type of a vector:

```
julia> typeof([3, 1, 4])              # 1-dimensional array of Int64s
Vector{Int64} (alias for Array{Int64, 1})
julia> typeof([3.1415, 1.618, 2.7182]) # 1-dimensional array of Float64s
Vector{Float64} (alias for Array{Float64, 1})
julia> Vector{Float64} # alias for a 1-dimensional array
Vector{Float64} (alias for Array{Float64, 1})
```

We index into vectors using square brackets:

```
julia> x[1]       # first element is indexed by 1
3.1415
julia> x[3]       # third element
2.7182
julia> x[end]     # use end to reference the end of the array
2.7182
julia> x[end-1]   # this returns the second to last element
1.618
```

We can pull out a range of elements from an array. Ranges are specified using a colon notation:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
julia> x =  [1, 2, 5, 3, 1]
5-element Vector{Int64}:
 1
 2
 5
 3
 1
julia> x[1:3]      # pull out the first three elements
3-element Vector{Int64}:
 1
 2
 5
julia> x[1:2:end]  # pull out every other element
3-element Vector{Int64}:
 1
 5
 1
julia> x[end:-1:1] # pull out all the elements in reverse order
5-element Vector{Int64}:
 1
 3
 5
 2
 1
```

We can perform a variety of operations on arrays. The exclamation mark at the end of function names is used to indicate that the function *mutates* (i.e., changes) the input:

```
julia> length(x)
5
julia> [x, x]             # vector of vectors
2-element Vector{Vector{Int64}}:
 [1, 2, 5, 3, 1]
 [1, 2, 5, 3, 1]
julia> push!(x, -1)       # add an element to the end
6-element Vector{Int64}:
  1
  2
  5
  3
  1
 -1
julia> pop!(x)            # remove an element from the end
-1
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
julia> append!(x, [2, 3]) # append [2, 3] to the end of x
7-element Vector{Int64}:
 1
 2
 5
 3
 1
 2
 3
julia> sort!(x)           # sort the elements, altering the same vector
7-element Vector{Int64}:
 1
 1
 2
 2
 3
 3
 5
julia> sort(x);           # sort the elements as a new vector
julia> x[1] = 2; print(x) # change the first element to 2
[2, 1, 2, 2, 3, 3, 5]
julia> x = [1, 2];
julia> y = [3, 4];
julia> x + y              # add vectors
2-element Vector{Int64}:
 4
 6
julia> 3x - [1, 2]        # multiply by a scalar and subtract
2-element Vector{Int64}:
 2
 4
julia> using LinearAlgebra
julia> dot(x, y)          # dot product available after using LinearAlgebra
11
julia> x⋅y                # dot product using unicode character, use \cdot[tab] in console
11
julia> prod(y)            # product of all the elements in y
12
```

It is often useful to apply various functions elementwise to vectors. This is a form of *broadcasting*. With infix operators (e.g., `+`, `*`, and `^`), a dot is prefixed to indicate elementwise broadcasting. With functions like `sqrt` and `sin`, the dot is postfixed:

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
julia> x .* y     # elementwise multiplication
2-element Vector{Int64}:
 3
 8
julia> x .^ 2     # elementwise squaring
2-element Vector{Int64}:
 1
 4
julia> sin.(x)    # elementwise application of sin
2-element Vector{Float64}:
 0.841471
 0.909297
julia> sqrt.(x)   # elementwise application of sqrt
2-element Vector{Float64}:
 1.0
 1.41421
julia> min.(x, 3) # elementwise min between x and a scalar
2-element Vector{Int64}:
 1
 2
julia> x .= y     # assign all elements of x to those in y
2-element Vector{Int64}:
 3
 4
```

Arrays are assigned by reference; they are not copied when assigned to a new variable. To make a copy, we use the `copy` function.

```
julia> x = [1, 2, 3];
julia> y = x;
julia> y[1] = 4;
julia> x[1]   # x is also changed
4
julia> y = copy(x);
julia> y[1] = 5;
julia> x[1]   # x is not changed
4
julia> x .= y # assign all elements of x to those in y
3-element Vector{Int64}:
 5
 2
 3
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A.1.5 Matrices

A *matrix* is a two-dimensional array. Like a vector, it is constructed using square brackets. We use spaces to delimit elements in the same row and semicolons to delimit rows. We can also index into the matrix and output submatrices using ranges. Some of the operations require the use of the built-in LinearAlgebra package:

```
julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
julia> typeof(X)  # a 2-dimensional array of Int64s
Matrix{Int64} (alias for Array{Int64, 2})
julia> X[2]       # second element using column-major ordering
4
julia> X[3,2]     # element in third row and second column
8
julia> X[1,:]     # extract the first row
3-element Vector{Int64}:
 1
 2
 3
julia> X[:,2]     # extract the second column
4-element Vector{Int64}:
  2
  5
  8
 11
julia> X[:,1:2]   # extract the first two columns
4×2 Matrix{Int64}:
  1   2
  4   5
  7   8
 10  11
julia> X[1:2,1:2] # extract a 2x2 submatrix from the top left of X
2×2 Matrix{Int64}:
 1  2
 4  5
julia> Matrix{Float64} # alias for a 2-dimensional array
Matrix{Float64} (alias for Array{Float64, 2})
```

We can also construct a variety of special matrices and use array comprehensions:

```
julia> Matrix(1.0I(3))                    # 3x3 identity matrix
3×3 Matrix{Float64}:
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
julia> Matrix(Diagonal([3, 2, 1]))         # 3x3 diagonal matrix with 3, 2, 1 on diagonal
3×3 Matrix{Int64}:
 3  0  0
 0  2  0
 0  0  1
julia> zeros(3,2)                          # 3x2 matrix of zeros
3×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
 0.0  0.0
julia> rand(3,2)                           # 3x2 random matrix
3×2 Matrix{Float64}:
 0.938064  0.122173
 0.49568   0.747459
 0.247823  0.43649
julia> [sin(x + y) for x in 1:3, y in 1:2] # array comprehension
3×2 Matrix{Float64}:
  0.909297   0.14112
  0.14112   -0.756802
 -0.756802  -0.958924
```

Matrix operations include the following:

```
julia> X'              # complex conjugate transpose
3×4 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7  10
 2  5  8  11
 3  6  9  12
julia> 3X .+ 2         # multiplying by scalar and adding scalar
4×3 Matrix{Int64}:
  5   8  11
 14  17  20
 23  26  29
 32  35  38
julia> X = [1 3; 3 1]; # create an invertible matrix
julia> inv(X)          # inversion
2×2 Matrix{Float64}:
 -0.125   0.375
  0.375  -0.125
julia> pinv(X)         # pseudoinverse
2×2 Matrix{Float64}:
 -0.125   0.375
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
  0.375  -0.125
julia> det(X)          # determinant
-8.0
julia> [X X]           # horizontal concatenation, same as hcat(X, X)
2×4 Matrix{Int64}:
 1  3  1  3
 3  1  3  1
julia> [X; X]          # vertical concatenation, same as vcat(X, X)
4×2 Matrix{Int64}:
 1  3
 3  1
 1  3
 3  1
julia> sin.(X)         # elementwise application of sin
2×2 Matrix{Float64}:
 0.841471  0.14112
 0.14112   0.841471
julia> map(sin, X)     # elementwise application of sin
2×2 Matrix{Float64}:
 0.841471  0.14112
 0.14112   0.841471
julia> vec(X)          # reshape an array as a vector
4-element Vector{Int64}:
 1
 3
 3
 1
```

The backslash operator \ is used to efficiently solve linear systems $\mathbf{Ax} = \mathbf{b}$ and associated least-squares problems:

```
julia> A = [1 0; 1 -2];     # define a full-rank square matrix
julia> b = [32, -4];        # define a vector
julia> x = A\b              # solve the linear system Ax = b
2-element Vector{Float64}:
 32.0
 18.0
julia> A*x - b              # verify the solution (should be close to zero)
2-element Vector{Float64}:
 0.0
 0.0
julia> A = [1 2; 3 4; 5 6]; # define matrix with more rows than columns
julia> b = [0, 8, 9];       # define a vector
julia> x = A\b              # solve the least-squares problem min_x ||Ax - b||_2
2-element Vector{Float64}:
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
  3.33333
 -1.08333
julia> norm(A*x - b)        # compute the residual
2.85774
julia> A = [1 2 3; 4 5 6];  # define a matrix with more columns than rows
julia> b = [7, 8];          # define a vector
julia> x = A\b              # solve the problem min_x ||x||_2 such that Ax = b
3-element Vector{Float64}:
 -3.05556
  0.111111
  3.27778
julia> A*x - b              # verify the solution (should be close to zero)
2-element Vector{Float64}:
 -1.77636e-15
  1.77636e-15
```

A.1.6 Tuples

A *tuple* is an ordered list of values, potentially of different types. They are constructed with parentheses. They are similar to vectors, but they cannot be mutated:

```
julia> x = ()   # the empty tuple
()
julia> isempty(x)
true
julia> x = (1,) # tuples of one element need the trailing comma
(1,)
julia> typeof(x)
Tuple{Int64}
julia> x = (1, 0, [1, 2],  2.5029, 4.6692) # third element is a vector
(1, 0, [1, 2], 2.5029, 4.6692)
julia> typeof(x)
Tuple{Int64, Int64, Vector{Int64}, Float64, Float64}
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
julia> a, b = x;
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
julia> a
1
julia> b
2
```

A.1.7 Named Tuples

A *named tuple* is like a tuple, but each entry has its own name:

```
julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a⇒10)
(a = 10,)
julia> (; :a⇒10, :b⇒11)
(a = 10, b = 11)
julia> merge(x, (d=3, e=10))  # merge two named tuples
(a = 1, b = -Inf, d = 3, e = 10)
```

A.1.8 Dictionaries

A *dictionary* is a collection of key-value pairs. Key-value pairs are indicated with a double arrow operator `=>`. We can index into a dictionary using square brackets, just as with arrays and tuples:

```
julia> x = Dict(); # empty dictionary
julia> x[3] = 4 # associate key 3 with value 4
4
julia> x = Dict(3⇒4, 5⇒1) # create a dictionary with two key-value pairs
Dict{Int64, Int64} with 2 entries:
  5 ⇒ 1
  3 ⇒ 4
julia> x[5]         # return the value associated with key 5
1
julia> haskey(x, 3) # check whether dictionary has key 3
true
julia> haskey(x, 4) # check whether dictionary has key 4
false
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A.1.9 Composite Types

A *composite type* is a collection of named fields. By default, an instance of a composite type is immutable (i.e., it cannot change). We use the `struct` keyword and then give the new type a name and list the names of the fields:

```
struct A
    a
    b
end
```

Adding the keyword `mutable` makes it so that an instance can change:

```
mutable struct B
    a
    b
end
```

Composite types are constructed using parentheses, between which we pass in values for each field:

```
x = A(1.414, 1.732)
```

We can extract the values of the fields using dot notation. For example, `x.a` returns the value of the field `a` in the composite type `x`.

The double-colon operator can be used to specify the type for any field:

```
struct A
    a::Int64
    b::Float64
end
```

These type annotations require that we pass in an `Int64` for the first field and a `Float64` for the second field. For compactness, this book does not use type annotations, but it is at the expense of performance. Type annotations allow Julia to improve runtime performance because the compiler can optimize the underlying code for specific types.

A.1.10 Abstract Types

So far we have discussed *concrete types*, which are types that we can construct. However, concrete types are only part of the type hierarchy. There are also *abstract types*, which are supertypes of concrete types and other abstract types.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

We can explore the type hierarchy of the `Float64` type shown in figure A.1 using the `supertype` and `subtypes` functions:

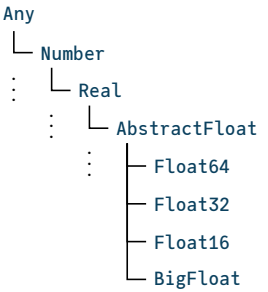


Figure A.1. The type hierarchy for the `Float64` type.

```
julia> supertype(Float64)
AbstractFloat
julia> supertype(AbstractFloat)
Real
julia> supertype(Real)
Number
julia> supertype(Number)
Any
julia> supertype(Any)          # Any is at the top of the hierarchy
Any
julia> using InteractiveUtils  # required for using subtypes in scripts
julia> subtypes(AbstractFloat) # different types of AbstractFloats
5-element Vector{Any}:
 BigFloat
 Core.BFloat16
 Float16
 Float32
 Float64
julia> subtypes(Float64)       # Float64 does not have any subtypes
Type[]
```

We can define our own abstract types:

```
abstract type C end
abstract type D <: C end # D is an abstract subtype of C
struct E <: D # E is a composite type that is a subtype of D
    a
end
```

A.1.11 Parametric Types

Julia supports *parametric types*, which are types that take parameters. The parameters to a parametric type are given within braces and delimited by commas. We have already seen a parametric type with our dictionary example:

```
julia> x = Dict(3⇒1.4, 1⇒5.9)
Dict{Int64, Float64} with 2 entries:
  3 ⇒ 1.4
  1 ⇒ 5.9
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

For dictionaries, the first parameter specifies the key type, and the second parameter specifies the value type. The example has `Int64` keys and `Float64` values, making the dictionary of type `Dict{Int64,Float64}`. Julia was able to infer these types based on the input, but we could have specified them explicitly:

```
julia> x = Dict{Int64,Float64}(3⇒1.4, 1⇒5.9);
```

While it is possible to define our own parametric types, we do not need to do so in this text.

A.2 Functions

A *function* maps its arguments, given as a tuple, to a result that is returned.

A.2.1 Named Functions

One way to define a *named function* is to use the `function` keyword, followed by the name of the function and a tuple of names of arguments:

```
function f(x, y)
    return x + y
end
```

We can also define functions compactly using assignment form:

```
julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415
```

A.2.2 Anonymous Functions

An *anonymous function* is not given a name, though it can be assigned to a named variable. One way to define an anonymous function is to use the arrow operator:

```
julia> h = x -> x^2 + 1 # assign anonymous function with input x to a variable h
#1 (generic function with 1 method)
julia> h(3)
10
julia> g(f, a, b) = [f(a), f(b)]; # applies function f to a and b and returns array
julia> g(h, 5, 10)
2-element Vector{Int64}:
  26
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
 101
julia> g(x→sin(x)+1, 10, 20)
2-element Vector{Float64}:
 0.4559788891106302
 1.9129452507276277
```

A.2.3 *Optional Arguments*

We can assign a default value to an argument, making the specification of that argument optional:

```
julia> f(x=10) = x^2;
julia> f()
100
julia> f(3)
9
julia> f(x, y, z=1) = x*y + z;
julia> f(1, 2, 3)
5
julia> f(1, 2)
3
```

A.2.4 *Keyword Arguments*

Functions may use *keyword arguments*, which are arguments that are named when the function is called. Keyword arguments are given after all the positional arguments. A semicolon is placed before any keywords, separating them from the other arguments:

```
julia> f(; x = 0) = x + 1;
julia> f()
1
julia> f(x = 10)
11
julia> f(x, y = 10; z = 2) = (x + y)*z;
julia> f(1)
22
julia> f(2, z = 3)
36
julia> f(2, 3)
10
julia> f(2, 3, z = 1)
5
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A.2.5 Dispatch

The types of the arguments passed to a function can be specified using the double colon operator. If multiple methods of the same function are provided, Julia will execute the appropriate method. The mechanism for choosing which method to execute is called *dispatch*:

```
julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.141500000000001
julia> f(1.3)
4.4415000000000004
```

The method with a type signature that best matches the types of the arguments given will be used:

```
julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415
```

A.2.6 Splatting

It is often useful to *splat* the elements of a vector or a tuple into the arguments to a function using the `...` operator:

```
julia> f(x,y,z) = x + y - z;
julia> a = [3, 1, 2];
julia> f(a...)
2
julia> b = (2, 2, 0);
julia> f(b...)
4
julia> c = ([0,0],[1,1]);
julia> f([2,2], c...)
2-element Vector{Int64}:
 1
 1
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

A.3 Control Flow

We can control the flow of our programs using conditional evaluation and loops. This section provides some of the syntax used in the book.

A.3.1 Conditional Evaluation

Conditional evaluation will check the value of a Boolean expression and then evaluate the appropriate block of code. One of the most common ways to do this is with an `if` statement:

```
if x < y
    # run this if x < y
elseif x > y
    # run this if x > y
else
    # run this if x == y
end
```

We can also use the *ternary operator* with its question mark and colon syntax. It checks the Boolean expression before the question mark. If the expression evaluates to true, then it returns what comes before the colon; otherwise, it returns what comes after the colon:

```
julia> f(x) = x > 0 ? x : 0;
julia> f(-10)
0
julia> f(10)
10
```

A.3.2 Loops

A *loop* allows for repeated evaluation of expressions. One type of loop is the while loop, which repeatedly evaluates a block of expressions as long as the specified condition after the `while` keyword is met. The following example sums the values in the array `X`:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while !isempty(X)
    s += pop!(X)
end
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Another type of loop is the for loop, which uses the `for` keyword. The following example will also sum over the values in the array `X` but will not modify `X`:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in X
    s += y
end
```

The `in` keyword can be replaced by `=` or `∈`. The following code block is equivalent:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(X)
    s += X[i]
end
```

A.3.3 Iterators

We can iterate over collections in contexts such as for loops and array comprehensions. To demonstrate various iterators, we will use the `collect` function, which returns an array of all items generated by an iterator:

```
julia> X = ["feed", "sing", "ignore"];
julia> collect(enumerate(X)) # return the count and the element
3-element Vector{Tuple{Int64, String}}:
 (1, "feed")
 (2, "sing")
 (3, "ignore")
julia> collect(eachindex(X)) # equivalent to 1:length(X)
3-element Vector{Int64}:
 1
 2
 3
julia> Y = [-5, -0.5, 0];
julia> collect(zip(X, Y))    # iterate over multiple iterators simultaneously
3-element Vector{Tuple{String, Float64}}:
 ("feed", -5.0)
 ("sing", -0.5)
 ("ignore", 0.0)
julia> import IterTools: subsets
julia> collect(subsets(X))   # iterate over all subsets
8-element Vector{Vector{String}}:
 []
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
 ["feed"]
 ["sing"]
 ["feed", "sing"]
 ["ignore"]
 ["feed", "ignore"]
 ["sing", "ignore"]
 ["feed", "sing", "ignore"]
julia> Z = [1 2; 3 4; 5 6];
julia> import Base.Iterators: product
julia> collect(product(X,Y)) # iterate over Cartesian product of multiple iterators
3×3 Matrix{Tuple{String, Float64}}:
 ("feed", -5.0)    ("feed", -0.5)    ("feed", 0.0)
 ("sing", -5.0)    ("sing", -0.5)    ("sing", 0.0)
 ("ignore", -5.0)  ("ignore", -0.5)  ("ignore", 0.0)
```

A.4 Packages

A *package* is a collection of Julia code and possibly other external libraries that can be imported to provide additional functionality. This section briefly reviews a few of the key packages that we build upon in this book. To add a registered package like `Distributions.jl`, we can run

```
using Pkg
Pkg.add("Distributions")
```

To update packages, we use

```
Pkg.update()
```

To use a package, we use the keyword `using` as follows:

```
using Distributions
```

Several code blocks in this text specify a package import with `using`. Some code blocks make use of functions that are not explicitly imported. For instance, the `var` function is provided by `Statistics.jl`, and the golden ratio φ is defined in `Base.MathConstants.jl`. Other excluded packages are `InteractiveUtils.jl`, `LinearAlgebra.jl`, `QuadGK.jl`, `Random.jl`, and `StatsBase.jl`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

B Test Functions

Researchers in optimization use several *test functions* to evaluate optimization algorithms. This appendix covers several test functions used throughout this book.

B.1 Ackley's Function

Ackley's function (figure B.1) is used to test a method's susceptibility to getting stuck in local minima. It comprises two primary components—a sinusoidal component that produces a multitude of local minima and an exponential bell curve centered at the origin, which establishes the function's global minimum.

Ackley's function is defined for any number of dimensions d:

$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1) \tag{B.1}$$

with a global minimum at the origin with an optimal value of zero. Typically, $a = 20$, $b = 0.2$, and $c = 2\pi$. Ackley's function is implemented in algorithm B.1.

B.2 Booth's Function

Booth's function (figure B.2) is a two-dimensional quadratic function.

Its equation is given by

$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \tag{B.2}$$

with a global minimum at $[1, 3]$ with an optimal value of zero. It is implemented in algorithm B.2.

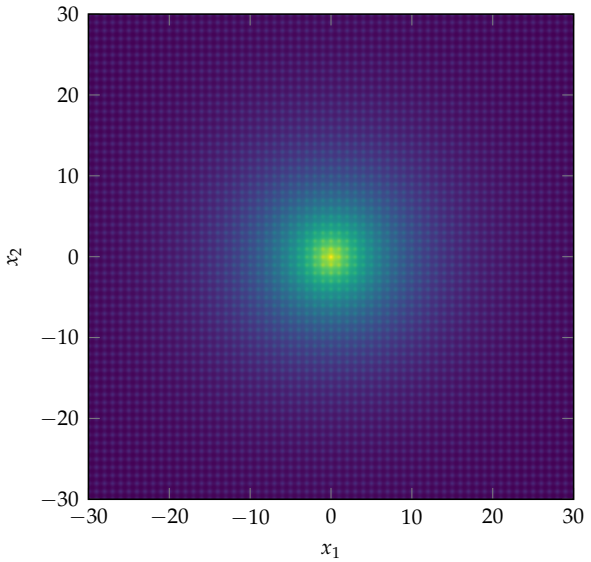


Figure B.1. The two-dimensional version of Ackley's function. The global minimum is at the origin.

```
function ackley(x, a=20, b=0.2, c=2π)
    d = length(x)
    return -a*exp(-b*sqrt(sum(x.^2)/d)) -
           exp(sum(cos.(c*xi) for xi in x)/d) + a + exp(1)
end
```

Algorithm B.1. Ackley's function with d-dimensional input vector `x` and three optional parameters.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

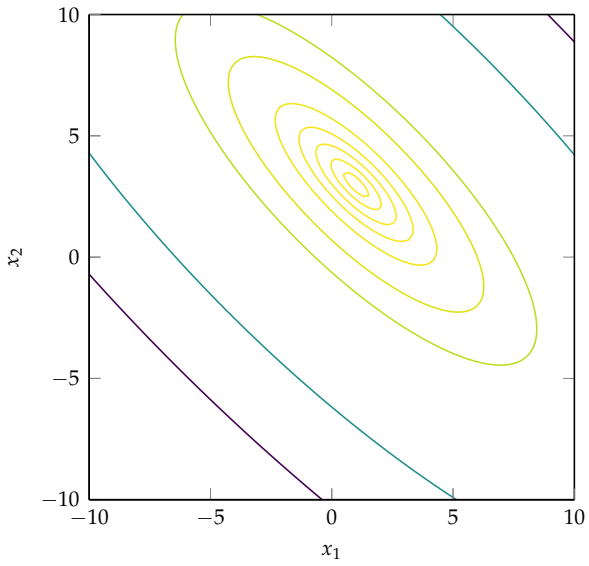


Figure B.2. Booth's function with a global minimum at $[1, 3]$.

```
booth(x) = (x[1]+2x[2]-7)^2 + (2x[1]+x[2]-5)^2
```

Algorithm B.2. Booth's function with two-dimensional input vector `x`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

B.3 Branin Function

The *Branin function* (figure B.3) is a two-dimensional function,

$$f(\mathbf{x}) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s \tag{B.3}$$

with recommended values $a = 1$, $b = 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = 10$, and $t = 1/(8\pi)$.

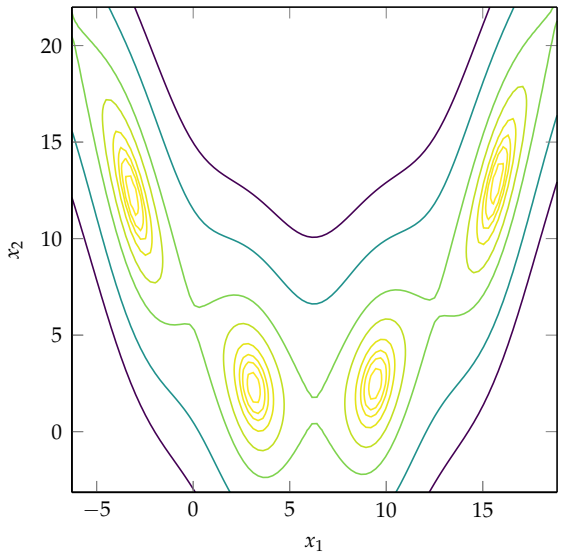


Figure B.3. The Branin function, with four global minima.

It has no local minima aside from global minima with $x_1 = \pi + 2\pi m$ for integral m. Four of these minima are:

$$\left\{ \begin{bmatrix} -\pi \\ 12.275 \end{bmatrix}, \begin{bmatrix} \pi \\ 2.275 \end{bmatrix}, \begin{bmatrix} 3\pi \\ 2.475 \end{bmatrix}, \begin{bmatrix} 5\pi \\ 12.875 \end{bmatrix} \right\} \tag{B.4}$$

with $f(\mathbf{x}^*) \approx 0.397887$. It is implemented in algorithm B.3.

```
function branin(x; a=1, b=5.1/(4π^2), c=5/π, r=6, s=10, t=1/(8π))
    return a*(x[2]-b*x[1]^2+c*x[1]-r)^2 + s*(1-t)*cos(x[1]) + s
end
```

Algorithm B.3. The Branin function with two-dimensional input vector `x` and six optional parameters.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

B.4 Flower Function

The *flower function* (figure B.4) is a two-dimensional test function whose contour plot has flower-like petals originating from the origin.

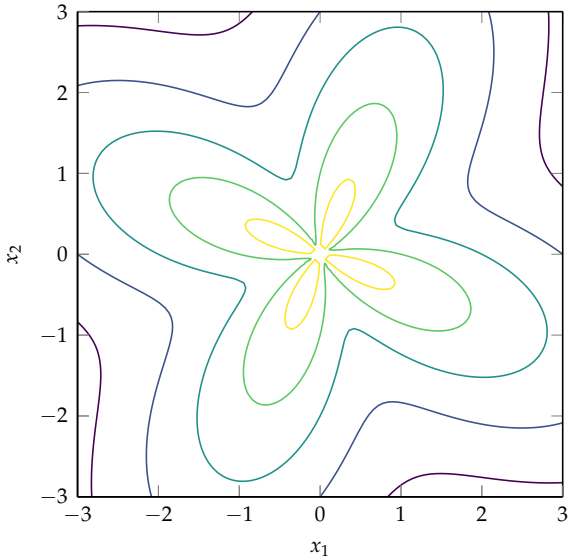


Figure B.4. The flower function.

The equation is

$$f(\mathbf{x}) = a\|\mathbf{x}\| + b\sin(c\tan^{-1}(x_2, x_1)) \tag{B.5}$$

with its parameters typically set to $a = 1$, $b = 1$, and $c = 4$.

The flower function is minimized near the origin but does not have a global minimum due to $\tan^{-1}$ being undefined at $[0, 0]$. It is implemented in algorithm B.4.

```
function flower(x; a=1, b=1, c=4)
    return a*norm(x) + b*sin(c*atan(x[2], x[1]))
end
```

Algorithm B.4. The flower function with two-dimensional input vector `x` and three optional parameters.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

B.5 Michalewicz Function

The *Michalewicz function* (figure B.5) is a d-dimensional optimization function with several steep valleys.

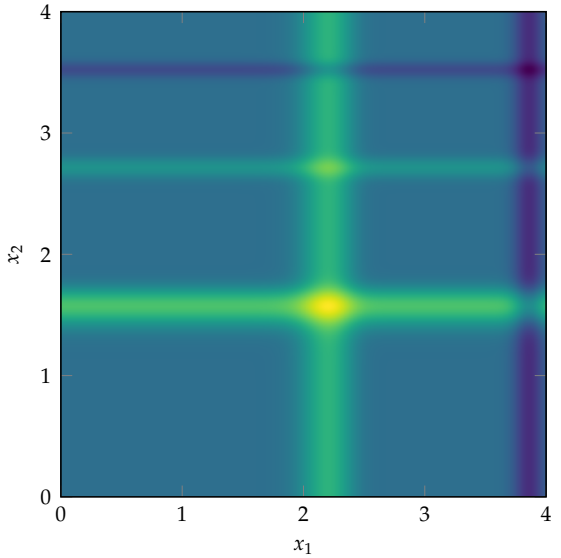


Figure B.5. The Michalewicz function.

Its equation is

$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{i x_i^2}{\pi}\right) \tag{B.6}$$

where the parameter m, typically 10, controls the steepness. The global minimum depends on the number of dimensions. In two dimensions the minimum is at approximately $[2.20, 1.57]$ with $f(\mathbf{x}^*) = -1.8013$. It is implemented in algorithm B.5.

```
function michalewicz(x; m=10)
    return -sum(sin(v)*sin(i*v^2/π)^(2m) for
                (i,v) in enumerate(x))
end
```

Algorithm B.5. The Michalewicz function with input vector `x` and optional steepness parameter `m`.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

B.6 Rosenbrock's Banana Function

The *Rosenbrock function* (figure B.6), also called Rosenbrock's valley or Rosenbrock's banana function, is a well-known unconstrained test function developed by Rosenbrock in 1960.[1] It has a global minimum inside a long, curved valley. Most optimization algorithms have no problem finding the valley but have difficulties traversing along the valley to the global minimum.

[1] H. H. Rosenbrock, "An Automatic Method for Finding the Greatest or Least Value of a Function," *The Computer Journal*, vol. 3, no. 3, pp. 175–184, 1960.

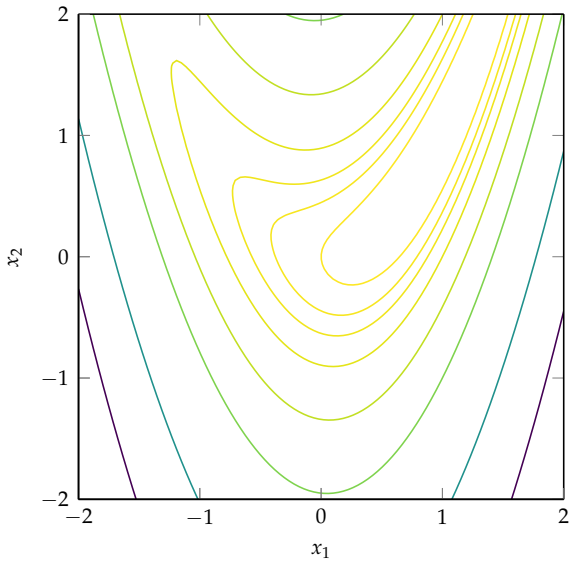


Figure B.6. The Rosenbrock function with $a = 1$ and $b = 5$. The global minimum is at $[1, 1]$.

The Rosenbrock function is

$$f(\mathbf{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2 \tag{B.7}$$

with a global minimum at $[a, a^2]$ at which $f(\mathbf{x}^*) = 0$. This text uses $a = 1$ and $b = 5$.

```
rosenbrock(x; a=1, b=5) = (a-x[1])^2 + b*(x[2] - x[1]^2)^2
```

Algorithm B.6. The Rosenbrock function with two-dimensional input vector `x` and two optional parameters.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

The Rosenbrock function can be extended to multiple dimensions. One common extension[2] for n dimensions is:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} (a - x_i)^2 + b(x_{i+1} - x_i^2)^2 \tag{B.8}$$

[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

The Rosenbrock function is implemented in algorithm B.6. The multidimensional Rosenbrock function is implemented in algorithm B.7.

```
function extended_rosenbrock(x; a=1, b=5)
    return sum((a .- x[1:end-1]).^2 .+ b*(x[2:end] .- x[1:end-1].^2).^2)
end
```

Algorithm B.7. A multidimensional extension to the Rosenbrock function with an n-dimensional input vector `x` for $n > 1$ and two optional parameters.

B.7 Wheeler's Ridge

Wheeler's ridge (figure B.7) is a two-dimensional function with a single global minimum in a deep curved peak. The function has two ridges, one along the positive and one along the negative first coordinate axis. A gradient descent method will diverge along the negative axis ridge. The function is very flat away from the optimum and the ridge.

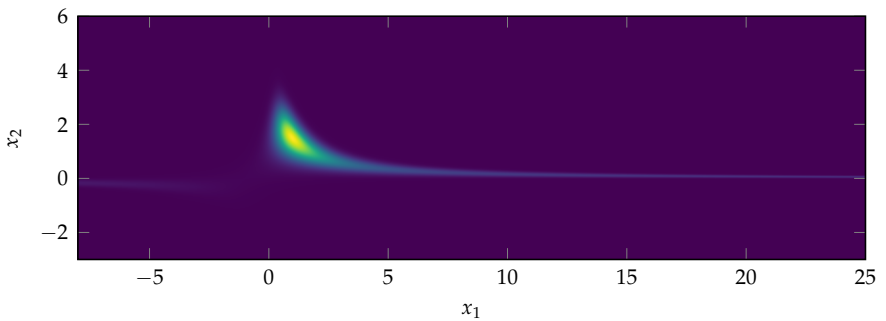


Figure B.7. Wheeler's ridge showing the two ridges and the peak containing the global minimum.

The function is given by

$$f(\mathbf{x}) = -\exp(-(x_1 x_2 - a)^2 - (x_2 - a)^2) \tag{B.9}$$

with a typically equal to 1.5, for which the global optimum of -1 is at $[1, 3/2]$.

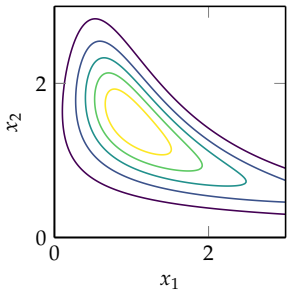


Figure B.8. A contour plot of the minimal region of Wheeler's ridge.

Wheeler's ridge has a smooth contour plot (figure B.8) when evaluated over $x_1 \in [0, 3]$ and $x_2 \in [0, 3]$. It is implemented in algorithm B.8.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
wheeler(x, a=1.5) = -exp(-(x[1]*x[2] - a)^2 -(x[2]-a)^2)
```

Algorithm B.8. Wheeler's ridge, which takes in a two-dimensional design point `x` and an optional scalar parameter `a`.

B.8 Circle Function

The *circle function* (algorithm B.9) is a simple multiobjective test function given by

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 1 - r\cos(\theta) \\ 1 - r\sin(\theta) \end{bmatrix} \tag{B.10}$$

where $\theta = x_1$ and r is obtained by passing x_2 through

$$r = \frac{1}{2} + \frac{1}{2}\left(\frac{2x_2}{1 + x_2^2}\right) \tag{B.11}$$

The Pareto frontier has $r = 1$ and $\theta \mod 2\pi \in [0, \pi/2]$.

```
function circle(x)
    θ = x[1]
    r = 0.5 + 0.5*(2x[2]/(1+x[2]^2))
    y1 = 1 - r*cos(θ)
    y2 = 1 - r*sin(θ)
    return [y1, y2]
end
```

Algorithm B.9. The circle function, which takes in a two-dimensional design point `x` and produces a two-dimensional objective value.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C Mathematical Concepts

This appendix covers mathematical concepts used in the derivation and analysis of optimization methods. These concepts are used throughout this book.

C.1 Asymptotic Notation

Asymptotic notation is often used to characterize the growth of a function. This notation is sometimes called *big-Oh notation*, since the letter O is used because the growth rate of a function is often called its *order*. This notation can be used to describe the error associated with a numerical method or the time or space complexity of an algorithm. This notation provides an upper bound on a function as its argument approaches a certain value.

Mathematically, if $f(x) = O(g(x))$ as $x \to a$, then the absolute value of $f(x)$ is bounded by the absolute value of $g(x)$ times some positive and finite c for values of x sufficiently close to a:

$$|f(x)| \leq c|g(x)| \quad \text{for } x \to a \tag{C.1}$$

Writing $f(x) = O(g(x))$ is a common abuse of the equals sign. For example, $x^2 = O(x^2)$ and $2x^2 = O(x^2)$, but, of course, $x^2 \neq 2x^2$. In some mathematical texts, $O(g(x))$ represents the set of all functions that do not grow faster than $g(x)$. One might write, for example, $5x^2 \in O(x^2)$. An example of asymptotic notation is given in example C.1.

If $f(x)$ is a *linear combination*[1] of terms, then $O(f)$ corresponds to the order of the fastest growing term. Example C.2 compares the orders of several terms.

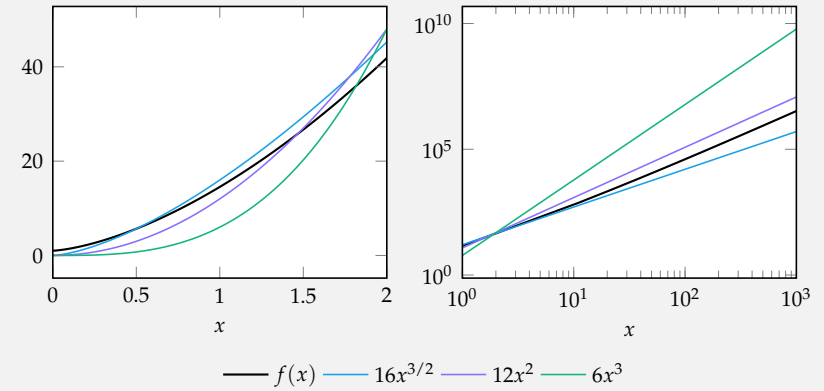
[1] A linear combination is a weighted sum of terms. If the terms are in a vector $\mathbf{x}$, then the linear combination is $w_1x_1 + w_2x_2 + \cdots = \mathbf{w}^\top\mathbf{x}$.

Example C.1. Asymptotic notation for a constant times a function.

Consider $f(x) = 10^6 e^x$ as $x \to \infty$. We want to find a c and a g such that $|f(x)| \leq c|g(x)|$. By choosing $c = 10^6$ and $g(x) = e^x$, we can satisfy this inequality as $x \to \infty$. Thus, $f = O(e^x)$ as $x \to \infty$.

Example C.2. An illustration of finding the order of a linear combination of terms.

Consider $f(x) = \cos(x) + x + 10x^{3/2} + 3x^2$. Here, f is a linear combination of terms. The terms $\cos(x)$, x, $x^{3/2}$, and x^2 grow at different rates as x approaches infinity. Below, we plot $f(x)$ along with $c|g(x)|$ for different functions $g(x)$ with c chosen such that $c|g(x)|$ exceeds $|f(x)|$ when $x = 2$.



There is no constant c such that $f(x)$ is always less than $c|x^{3/2}|$ for sufficiently large values of x. The same is true for $\cos(x)$ and x. We do find that there are values for c such that $c|x^2|$ and $c|x^3|$ are always greater than $f(x)$ for sufficiently large x. Hence, $f(x) = O(x^3)$, and in general $f(x) = O(x^m)$ for $m \geq 2$, along with other function classes like $f(x) = O(e^x)$. We typically discuss the order that provides the tightest upper bound. Thus, $f = O(x^2)$ as $x \to \infty$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C.2 Taylor Expansion

The *Taylor expansion*, also called the *Taylor series*, of a function is critical to understanding many of the optimization methods covered in this book, so we derive it here.

From the *first fundamental theorem of calculus*,[2] we know that

$$f(x+h) = f(x) + \int_0^h f'(x+a)\,da \tag{C.2}$$

[2] The first fundamental theorem of calculus relates a function to the integral of its derivative:

$$f(b) - f(a) = \int_a^b f'(x)\,dx$$

Nesting this definition produces the Taylor expansion of f about x:

$$\begin{aligned}
f(x+h) &= f(x) + \int_0^h \left(f'(x) + \int_0^a f''(x+b)\,db \right) da && \text{(C.3)}\\
&= f(x) + f'(x)h + \int_0^h \int_0^a f''(x+b)\,db\,da && \text{(C.4)}\\
&= f(x) + f'(x)h + \int_0^h \int_0^a \left(f''(x) + \int_0^b f'''(x+c)\,dc \right) db\,da && \text{(C.5)}\\
&= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \int_0^h \int_0^a \int_0^b f'''(x+c)\,dc\,db\,da && \text{(C.6)}\\
&\quad\vdots && \text{(C.7)}\\
&= f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \ldots && \text{(C.8)}\\
&= \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!}h^n && \text{(C.9)}
\end{aligned}$$

In some contexts, it may be more convenient to write the Taylor expansion of $f(x)$ about a point x_0 such that it is a function solely of x instead of $f(x+h)$:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \tag{C.10}$$

The Taylor expansion represents a function as an infinite sum of polynomial terms based on repeated derivatives at a single point. Any analytic function can be represented by its Taylor expansion within a local neighborhood.

A function can be locally approximated by using the first few terms of the Taylor expansion. Figure C.1 shows increasingly better approximations for $\cos(x)$ about $x = 1$. Including more terms increases the accuracy of the local approximation, but error still accumulates as one moves away from the expansion point.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

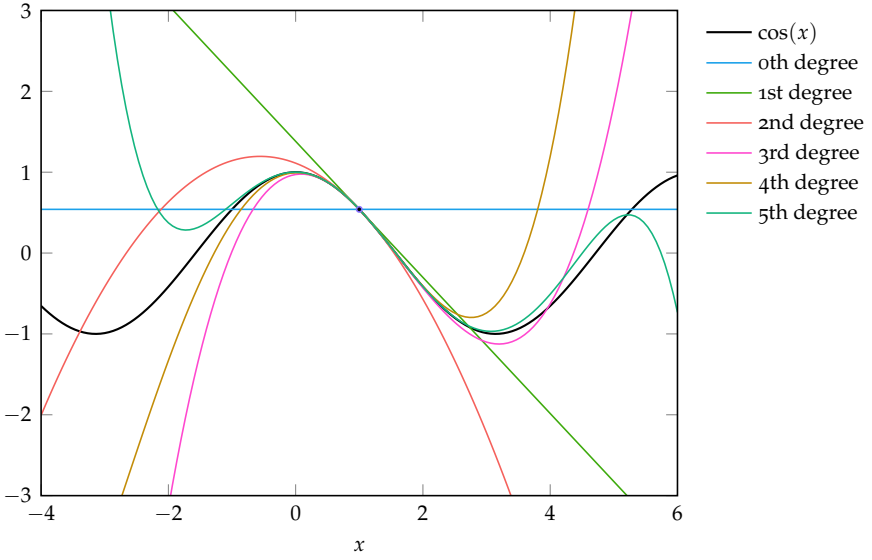


Figure C.1. Successive approximations of $\cos(x)$ about 1 based on the first n terms of the Taylor expansion.

A linear *Taylor approximation* uses the first two terms of the Taylor expansion:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) \tag{C.11}$$

A quadratic Taylor approximation uses the first three terms:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 \tag{C.12}$$

and so on.

In multiple dimensions, the Taylor expansion about $\mathbf{x}_0$ generalizes to

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top \nabla^2 f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \ldots \tag{C.13}$$

The first two terms form the tangent plane at $\mathbf{x}_0$. The third term incorporates local curvature. This text will use only the first three terms shown here.

C.3 Convexity

A *convex combination* of two vectors $\mathbf{x}$ and $\mathbf{y}$ is the result of

$$\alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \tag{C.14}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

for some $\alpha \in [0, 1]$. Convex combinations can be made from m vectors,

$$w_1\mathbf{v}^{(1)} + w_2\mathbf{v}^{(2)} + \cdots + w_m\mathbf{v}^{(m)} \tag{C.15}$$

with nonnegative weights $\mathbf{w}$ that sum to one.

A *convex set* is a set for which a line drawn between any two points in the set is entirely within the set. Mathematically, a set $\mathcal{S}$ is convex if we have

$$\alpha\mathbf{x} + (1 - \alpha)\mathbf{y} \in \mathcal{S} \tag{C.16}$$

for all $\mathbf{x}, \mathbf{y}$ in $\mathcal{S}$ and for all α in $[0, 1]$. A convex and a nonconvex set are shown in figure C.2.



A convex set

Not a convex set

Figure C.2. Convex and nonconvex sets.

A *convex function* is a *bowl-shaped* function whose domain is a convex set. By bowl-shaped, we mean it is a function such that any line drawn between two points in its domain does not lie below the function. A function f is convex over a convex set $\mathcal{S}$ if, for all $\mathbf{x}, \mathbf{y}$ in $\mathcal{S}$ and for all α in $[0, 1]$,

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) \leq \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y}) \tag{C.17}$$

Convex and concave regions of a function are shown in figure C.3.

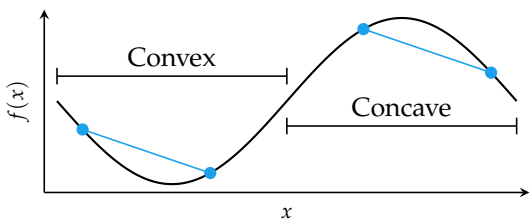


Figure C.3. Convex and nonconvex portions of a function.

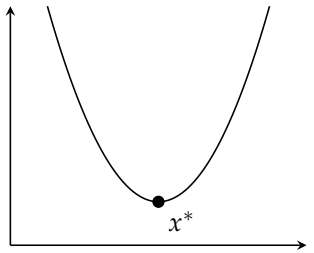
A function f is *strictly convex* over a convex set $\mathcal{S}$ if, for all $\mathbf{x}, \mathbf{y}$ in $\mathcal{S}$ and α in $(0, 1)$,

$$f(\alpha\mathbf{x} + (1 - \alpha)\mathbf{y}) < \alpha f(\mathbf{x}) + (1 - \alpha)f(\mathbf{y}) \tag{C.18}$$

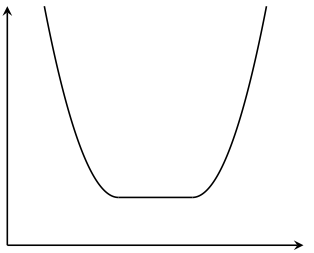
© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Strictly convex functions have at most one minimum, whereas a convex function can have flat regions.[3] Examples of strict and nonstrict convexity are shown in figure C.4.

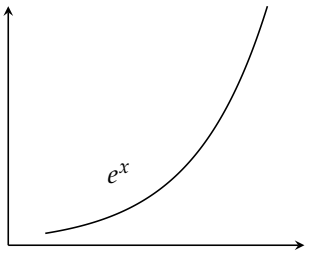
[3] Optimization of convex functions is the subject of the textbook by S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.



Strictly convex function with one global minimum.



Convex function without a unique global minimum.

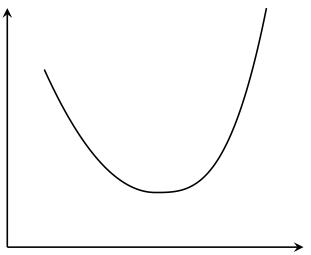


Strictly convex function without a global minimum.

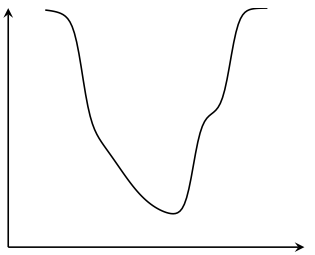
Figure C.4. Not all convex functions have single global minima.

A function f is *concave* if $-f$ is convex. Furthermore, f is *strictly concave* if $-f$ is strictly convex.

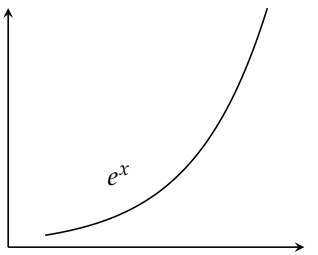
Not all convex functions are unimodal (defined in section 3.1) and not all unimodal functions are convex, as shown in figure C.5.



Unimodal and convex



Unimodal but nonconvex



Convex but nonunimodal

Figure C.5. Convexity and unimodality are not the same thing.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C.4 Norms

A *norm* is a function that assigns a length to a vector. To compute the distance between two vectors, we evaluate the norm of the difference between those two vectors. For example, the distance between points $\mathbf{a}$ and $\mathbf{b}$ using the *Euclidean norm* is

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2} \tag{C.19}$$

A function f is a norm if[4]

1. $f(\mathbf{x}) = 0$ if and only if $\mathbf{x}$ is the zero vector.
2. $f(a\mathbf{x}) = |a|f(\mathbf{x})$, such that lengths scale.
3. $f(\mathbf{a} + \mathbf{b}) \leq f(\mathbf{a}) + f(\mathbf{b})$, also known as the *triangle inequality*.

[4] Some properties that follow from these axioms include:

$$f(-\mathbf{x}) = f(\mathbf{x})$$
$$f(\mathbf{x}) \geq 0$$

The L_p norms are a commonly used set of norms parameterized by a scalar $p \geq 1$. The Euclidean norm in equation (C.19) is the L_2 norm. Several L_p norms are shown in table C.1.

The L_p norms are defined according to:

$$\|\mathbf{x}\|_p = \lim_{\rho \to p} (|x_1|^\rho + |x_2|^\rho + \cdots + |x_n|^\rho)^{\frac{1}{\rho}} \tag{C.20}$$

where the limit is necessary for defining the infinity norm, L_∞.[5]

[5] The L_∞ norm is also referred to as the *max norm*, *Chebyshev distance*, or *chessboard distance*. The latter name comes from the minimum number of moves a chess king needs to move between two chess squares.

C.5 Generalized Gradient

Gradients are not defined at nonsmooth points. The *generalized gradient*[6] is an extension of gradients to nonsmooth, but locally Lipschitz continuous, functions, generalizing the concept of steepest ascent and optimality conditions when the gradient is not defined.

[6] Also called the *Clarke generalized gradient*. F. H. Clarke, "Generalized Gradients and Applications," *Transactions of the American Mathematical Society*, vol. 205, pp. 247–262, 1975.

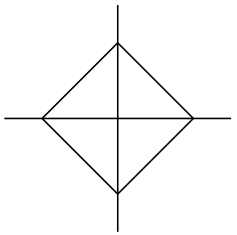
Before defining the generalized gradient, we first introduce the *Clarke directional derivative* $f^\circ_{\mathbf{s}}(\mathbf{x})$, defined as the upper limit of directional derivatives of f along a vector $\mathbf{s}$, evaluated at points approaching $\mathbf{x}$:[7]

$$f^\circ_{\mathbf{s}}(\mathbf{x}) = \limsup_{\mathbf{x}' \to \mathbf{x},\, h \downarrow 0} \frac{f(\mathbf{x}' + h\mathbf{s}) - f(\mathbf{x}')}{h} \tag{C.21}$$

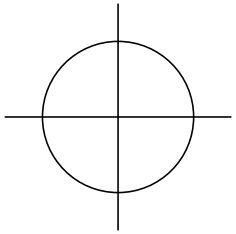
[7] The term lim sup denotes the *limit supremum*, meaning the least upper bound of all possible limit points.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

L_1: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$



L_2: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$



L_∞: $\|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, \cdots, |x_n|)$

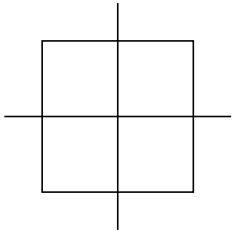


Table C.1. Common L_p norms. The illustrations show the shape of the norm contours in two dimensions. All points on the contour are equidistant from the origin under that norm.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Recall that the directional derivative can be written $\nabla_{\mathbf{s}} f(\mathbf{x}) = \nabla f(\mathbf{x})^\top \mathbf{s}$ when f is differentiable at $\mathbf{x}$. Regardless of whether f is differentiable at $\mathbf{x}$, we can define a generalized gradient to be any vector $\mathbf{v}$ such that $\mathbf{v}^\top \mathbf{s}$ does not exceed the Clarke directional derivative in any direction. The *Clarke subdifferential* $\partial^\circ f(\mathbf{x})$ of a locally Lipschitz continuous function f at a point $\mathbf{x}$ is the set of generalized gradients:[8]

$$\partial^\circ f(\mathbf{x}) = \left\{ \mathbf{v} \in \mathbb{R}^n \mid \mathbf{v}^\top \mathbf{s} \leq f'_{\mathbf{s}}(\mathbf{x}) \text{ for all } \mathbf{s} \in \mathbb{R}^n \right\} \tag{C.22}$$

[8] It is sufficient to use all unit vectors in $\mathbb{R}^n$ instead of $\mathbb{R}^n$ because positive scalings of $\mathbf{s}$ scale the Clarke directional derivative, and the zero vector is trivially satisfied.

This Clarke subdifferential is a nonempty, convex, compact set of vectors. Any generalized gradient in the set can be used in place of a gradient in a descent method, though the direction opposite of a generalized gradient is not necessarily a descent direction that leads to a local decrease in the function value. The subdifferential reduces to $\{\nabla f(\mathbf{x})\}$ when f is strictly differentiable[9] at $\mathbf{x}$. Example C.3 derives the Clarke subdifferential for a simple function.

[9] Strict differentiability requires that the slope converge as two independent points approach the fixed point. Strict differentiability at a point implies that the function is locally Lipschitz continuous at that point, which standard differentiability does not guarantee.

For a point $\mathbf{x}$ to be a local minimum of a locally Lipschitz function f, it is necessary that the Clarke subdifferential at $\mathbf{x}$ contain the zero vector. If f is convex, this condition is also sufficient.

The *subgradient* is a closely related concept that applies instead to convex functions.[10] Convex functions can always be bounded below by a supporting hyperplane. A subgradient $\mathbf{v}$ at a point $\mathbf{x}^{(0)}$ defines such a supporting hyperplane:

$$f(\mathbf{x}) \geq f(\mathbf{x}^{(0)}) + \mathbf{v}^\top \left(\mathbf{x} - \mathbf{x}^{(0)} \right) \tag{C.23}$$

[10] For an overview, see for example J.-B. Hiriart-Urruty and C. Lemaréchal, *Fundamentals of Convex Analysis*. Springer, 2001.

and the *subdifferential* is the set of all subgradients at $\mathbf{x}^{(0)}$. Any subgradient in the set can also be used in place of a gradient in a descent method, though it may not always lead to a local decrease in the objective function.

C.6 Matrix Calculus

This section derives two common gradients: $\nabla_{\mathbf{x}} \mathbf{b}^\top \mathbf{x}$ and $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x}$.

To obtain $\nabla_{\mathbf{x}} \mathbf{b}^\top \mathbf{x}$, we first expand the dot product:

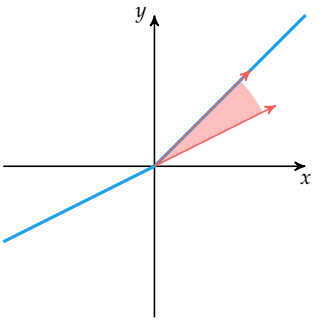
$$\mathbf{b}^\top \mathbf{x} = [b_1 x_1 + b_2 x_2 + \cdots + b_n x_n] \tag{C.24}$$

The partial derivative with respect to a single coordinate is:

$$\frac{\partial}{\partial x_i} \mathbf{b}^\top \mathbf{x} = b_i \tag{C.25}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Example C.3. Working out the generalized gradient on an example function. The red region in the figure below represents the set of slopes in the generalized gradient $\partial^\circ f(0)$.



Consider the function:

$$f(x) = \max(x/2, x) = \begin{cases} x/2 & \text{for } x \leq 0 \\ x & \text{for } x \geq 0 \end{cases}$$

This function has a derivative of $1/2$ for $x < 0$ and a derivative of 1 for $x > 0$. The derivative at $x = 0$ does not exist.

The Clarke directional derivative along $s = 1$ is:

$$f_1^\circ(0) = \limsup_{x' \to 0,\, h \downarrow 0} \frac{f(x'+h) - f(x')}{h}$$

Approaching from the right, $x' \geq 0$ and $x' + h \geq 0$, we get

$$\frac{x' + h - x'}{h} = 1$$

Approaching from the left, $x' \leq 0$ and $x' + h \leq 0$, we get

$$\frac{(x'+h)/2 - x'/2}{h} = \frac{1}{2}$$

Approaching from both sides, $x' < 0$ and $x' + h > 0$, we get

$$\frac{x' + h - x'/2}{h} = 1 + \frac{1}{2}\frac{x'}{h} \in \left(\frac{1}{2}, 1\right)$$

wherein we substituted $\frac{x'}{h} \in (-1, 0)$, derived using $x' + h > 0$, $-x' > 0$, and $h > 0$.

The limit supremum takes the least upper bound among all limits produced by different ways of letting $x' \to 0$ and $h \downarrow 0$, yielding $f_1^\circ(0) = 1$. Similarly, $f_{-1}^\circ(0) = \max(-1/2, -1) = -1/2$.

The generalized gradient at $x = 0$ is the set:

$$\begin{aligned} \partial^\circ f(0) &= \{v \in \mathbb{R} \mid vs \leq f_s^\circ(0) \text{ for all } s \in \{-1, 1\}\} \\ &= \{v \in \mathbb{R} \mid v \leq f_1^\circ(0) \text{ and } -v \leq f_{-1}^\circ(0)\} \\ &= \{v \in \mathbb{R} \mid v \leq 1 \text{ and } -v \leq -1/2\} \\ &= [1/2, 1] \end{aligned}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Thus, the gradient is:

$$\nabla_{\mathbf{x}} \mathbf{b}^\top \mathbf{x} = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{b} = \mathbf{b} \tag{C.26}$$

To obtain $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x}$ for a square matrix $\mathbf{A}$, we first expand $\mathbf{x}^\top \mathbf{A}\mathbf{x}$:

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^\top \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{C.27}$$

$$= \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^\top \begin{bmatrix} x_1 a_{11} + x_2 a_{12} + \cdots + x_n a_{1n} \\ x_1 a_{21} + x_2 a_{22} + \cdots + x_n a_{2n} \\ \vdots \\ x_1 a_{n1} + x_2 a_{n2} + \cdots + x_n a_{nn} \end{bmatrix} \tag{C.28}$$

$$= \begin{array}{c} x_1^2 a_{11} + x_1 x_2 a_{12} + \cdots + x_1 x_n a_{1n} + \\ x_1 x_2 a_{21} + x_2^2 a_{22} + \cdots + x_2 x_n a_{2n} + \\ \vdots \\ x_1 x_n a_{n1} + x_2 x_n a_{n2} + \cdots + x_n^2 a_{nn} \end{array} \tag{C.29}$$

The partial derivative with respect to the ith component is

$$\frac{\partial}{\partial x_i} \mathbf{x}^\top \mathbf{A}\mathbf{x} = \sum_{j=1}^{n} x_j \left(a_{ij} + a_{ji} \right) \tag{C.30}$$

The gradient is thus:

$$\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = \begin{bmatrix} \sum_{j=1}^{n} x_j \left(a_{1j} + a_{j1} \right) \\ \sum_{j=1}^{n} x_j \left(a_{2j} + a_{j2} \right) \\ \vdots \\ \sum_{j=1}^{n} x_j \left(a_{nj} + a_{jn} \right) \end{bmatrix} \tag{C.31}$$

$$= \begin{bmatrix} a_{11} + a_{11} & a_{12} + a_{21} & \cdots & a_{1n} + a_{n1} \\ a_{21} + a_{12} & a_{22} + a_{22} & \cdots & a_{2n} + a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} + a_{1n} & a_{n2} + a_{2n} & \cdots & a_{nn} + a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{C.32}$$

$$= \left(\mathbf{A} + \mathbf{A}^\top \right) \mathbf{x} \tag{C.33}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C.7 Positive Definiteness

The *positive definiteness* of matrices arises in linear algebra and optimization for a variety of reasons. A symmetric matrix $\mathbf{A}$ is *positive definite* if $\mathbf{x}^\top \mathbf{A}\mathbf{x}$ is positive for all points other than the origin: $\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$. A symmetric matrix $\mathbf{A}$ is *positive semidefinite* if $\mathbf{x}^\top \mathbf{A}\mathbf{x}$ is always nonnegative: $\mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x}$. If all eigenvalues of $\mathbf{A}$ are strictly positive, then $\mathbf{A}$ is positive definite and automatically also positive semidefinite. If all eigenvalues are nonnegative and some are zero, it is positive semidefinite but not positive definite.

Positive definiteness is useful in optimization because it guarantees that a function has a unique global minimum under certain conditions. For example, if the matrix $\mathbf{A}$ is positive definite in the function $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x}$, then f has a unique global minimum. If f is instead simply a twice-differentiable function, then the second-order Taylor approximation of f at $\mathbf{x}_0$ is

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H}_0 (\mathbf{x} - \mathbf{x}_0) \tag{C.34}$$

where $\mathbf{H}_0$ is the Hessian of f evaluated at $\mathbf{x}_0$. Knowing whether $(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H}_0 (\mathbf{x} - \mathbf{x}_0)$ has a unique global minimum is sufficient to determine whether the quadratic approximation has a unique global minimum.[11]

[11] The component $f(\mathbf{x}_0)$ merely shifts the function vertically. The component $\nabla f(\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0)$ is a linear term that is dominated by the quadratic term.

C.8 Matrix Decompositions

A *matrix decomposition* or *matrix factorization* is the decomposition of a matrix into a product of matrices. The component matrices making up this product often have known useful properties. There are many types of matrix decompositions. This section reviews a few used in this textbook assuming an $m \times n$ matrix $\mathbf{A}$ with real components.[12]

[12] Many of these decompositions extend to complex matrices as well. A review of matrix factorizations is provided by T. Lyche, *Numerical Linear Algebra and Matrix Factorizations*. Springer, 2020.

C.8.1 Cholesky Decomposition

A *Cholesky decomposition* of $\mathbf{A}$ satisfies

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\top \tag{C.35}$$

where $\mathbf{L}$ is a lower triangular matrix with positive diagonal entries. This decomposition is unique when $\mathbf{A}$ is positive definite.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ for a positive definite $\mathbf{A}$ typically takes $O(n^3)$ operations. If the Cholesky decomposition is known, then solving $\mathbf{LL}^\top \mathbf{x} = \mathbf{b}$ can be done by first solving $\mathbf{Ly} = \mathbf{b}$ with forward substitution in $O(n^2)$ and then solving $\mathbf{L}^\top \mathbf{x} = \mathbf{y}$ with back substitution in $O(n^2)$ time. While computing the Cholesky decomposition itself generally takes $O(n^3)$, it can be computed once for $\mathbf{A}$ and then reused to solve $\mathbf{Ax} = \mathbf{b}$ for multiple values of $\mathbf{b}$.

In Julia, the Cholesky decomposition is provided by the `cholesky` method in the package `LinearAlgebra.jl`:

```
julia> A = [21.0  -5.0  12.0; -5.0  13.0  -6.0; 12.0  -6.0   9.0];
julia> d = cholesky(A);
julia> d.L
3×3 LinearAlgebra.LowerTriangular{Float64, Matrix{Float64}}:
  4.58258    •         •
 -1.09109   3.4365     •
  2.61861  -0.914552  1.143
julia> d.L*d.L'
3×3 Matrix{Float64}:
 21.0  -5.0  12.0
 -5.0  13.0  -6.0
 12.0  -6.0   9.0
```

The `cholesky` method also provides an upper triangular matrix U such that $\mathbf{A} = \mathbf{U}^\top \mathbf{U}$:

```
julia> d.U
3×3 LinearAlgebra.UpperTriangular{Float64, Matrix{Float64}}:
 4.58258  -1.09109   2.61861
  •        3.4365   -0.914552
  •         •        1.143
julia> d.U'*d.U
3×3 Matrix{Float64}:
 21.0  -5.0  12.0
 -5.0  13.0  -6.0
 12.0  -6.0   9.0
```

C.8.2 LDL Decomposition

An *LDL decomposition* modifies the Cholesky decomposition to incorporate an additional diagonal matrix $\mathbf{D}$ such that:

$$\mathbf{A} = \mathbf{LDL}^\top \tag{C.36}$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

and $\mathbf{L}$ is a lower triangular matrix whose diagonal entries are all 1. This decomposition is sometimes called the *square-root-free Cholesky decomposition* because it avoids computing square roots.

We can recover a Cholesky decomposition from this decomposition through

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^\top = \mathbf{L}\mathbf{D}^{1/2}\left(\mathbf{D}^{1/2}\right)^\top \mathbf{L}^\top = \left(\mathbf{L}\mathbf{D}^{1/2}\right)\left(\mathbf{L}\mathbf{D}^{1/2}\right)^\top \tag{C.37}$$

In Julia, the LDL decomposition is provided by the `ldlt` method in the package `LinearAlgebra.jl`. The implementation requires that $\mathbf{A}$ be tridiagonal.

C.8.3 LQ Decomposition

An *LQ decomposition* of $\mathbf{A}$ satisfies[13]

$$\mathbf{A} = \mathbf{L}\mathbf{Q} \tag{C.38}$$

for an $m \times n$ lower triangular $\mathbf{L}$ and an $n \times n$ orthogonal matrix $\mathbf{Q}$.

[13] The *QR decomposition* is related, which is $\mathbf{A} = \mathbf{Q}\mathbf{R}$ for an $m \times m$ orthogonal $\mathbf{Q}$ and an $m \times n$ upper triangular $\mathbf{R}$.

An *orthogonal matrix*, or *orthonormal matrix*, satisfies $\mathbf{Q}^\top\mathbf{Q} = \mathbf{Q}\mathbf{Q}^\top = \mathbf{I}$. As such, the transpose of an orthogonal matrix is also its inverse, $\mathbf{Q}^\top = \mathbf{Q}^{-1}$. Multiplication by an orthogonal matrix also preserves the dot product:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top\mathbf{v} = \mathbf{u}^\top\mathbf{Q}^\top\mathbf{Q}\mathbf{v} = (\mathbf{Q}\mathbf{u})^\top(\mathbf{Q}\mathbf{v}) \tag{C.39}$$

In Julia, the LQ decomposition is provided by the `lq` method in the package `LinearAlgebra.jl`:

```
julia> d = lq(A)
LinearAlgebra.LQ{Float64, Matrix{Float64}, Vector{Float64}}
L factor:
3×3 Matrix{Float64}:
 -24.6982     0.0      0.0
   9.79829  -11.5756   0.0
 -15.7906     3.22047  1.13328
Q factor: 3×3 LinearAlgebra.LQPackedQ{Float64, Matrix{Float64}, Vector{Float64}}
julia> d.L * d.Q
3×3 Matrix{Float64}:
 21.0  -5.0  12.0
 -5.0  13.0  -6.0
 12.0  -6.0   9.0
```

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C.8.4 Singular Value Decomposition

A *singular value decomposition* of an $m \times n$ matrix $\mathbf{A}$ satisfies

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^\top \tag{C.40}$$

where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{S}$ is an $m \times n$ nonnegative diagonal matrix, and $\mathbf{V}$ is an $n \times n$ orthogonal matrix. Singular value decomposition is useful in a wide range of applications, including least-squares fitting, computing the pseudoinverse, identifying the rank of a matrix, and dimensionality reduction.

The columns of $\mathbf{U}$ are the eigenvectors of $\mathbf{A}\mathbf{A}^\top$, whereas the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{A}^\top\mathbf{A}$. The diagonal entries of $\mathbf{S}$, called the *singular values* of $\mathbf{A}$, are the square roots of the eigenvalues of $\mathbf{A}\mathbf{A}^\top$ and $\mathbf{A}^\top\mathbf{A}$. These singular values are typically arranged in decreasing order. If $\mathbf{A}$ has rank k, then k of the singular values are nonzero.

In Julia, the singular value decomposition is provided by the `svd`[14] method in the package `LinearAlgebra.jl`:

```
julia> d = svd(A)
LinearAlgebra.SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
3×3 Matrix{Float64}:
 -0.771231  -0.445746   -0.454438
  0.376402  -0.89506     0.239144
 -0.513346   0.0133833   0.858077
singular values:
3-element Vector{Float64}:
 31.4277
 10.5997
  0.972612
Vt factor:
3×3 Matrix{Float64}:
 -0.771231   0.376402  -0.513346
 -0.445746  -0.89506    0.0133833
 -0.454438   0.239144   0.858077
julia> d.U * Diagonal(d.S) * d.Vt
3×3 Matrix{Float64}:
 21.0  -5.0  12.0
 -5.0  13.0  -6.0
 12.0  -6.0   9.0
```

[14] By default, `svd` produces a 'thin' decomposition containing only the $\min(m, n)$ singular values. In this case, $\mathbf{U}$ is $m \times \min(m, n)$ and $\mathbf{V}$ is $n \times \min(m, n)$. We can call `svd(A, full=true)` to obtain a full singular value decomposition.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

C.9 Gaussian Distribution

The probability density function for a *univariate Gaussian*,[15] also called the *normal distribution*, is:

$$\mathcal{N}(x \mid \mu, \nu) = \frac{1}{\sqrt{2\pi\nu}} e^{-\frac{(x-\mu)^2}{2\nu}} \tag{C.41}$$

where μ is the mean and ν is the variance. The standard deviation is $\sigma = \sqrt{\nu}$. This distribution is plotted in figure C.6.

[15] The multivariate Gaussian is discussed in chapter 8 and chapter 18. The univariate Gaussian is used throughout.

The *cumulative distribution function* of a distribution maps x to the probability that drawing a value from that distribution will produce a value less than or equal to x. For a univariate Gaussian, the cumulative distribution function is given by

$$\Phi(x) \equiv \frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \tag{C.42}$$

where erf is the *error function*:

$$\text{erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-\tau^2} d\tau \tag{C.43}$$

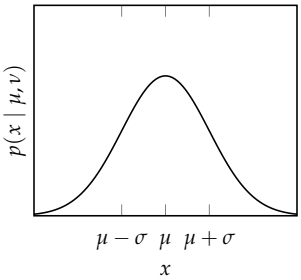


Figure C.6. A univariate Gaussian distribution, $\mathcal{N}(\mu, \nu)$.

C.10 Gaussian Quadrature

Gaussian quadrature is a technique for approximating integrals using a weighted sum of function evaluations.[16] The general form of the approximation is

$$\int_a^b p(x) f(x)\, dx \approx \sum_{i=1}^{m} w_i f(x_i) \tag{C.44}$$

where $p(x)$ is a known nonnegative weight function[17] over the finite or infinite interval $[a, b]$.

An m-point *quadrature rule* is a unique choice of points $x_i \in (a, b)$ and weights $w_i > 0$ for $i \in \{1, \ldots, m\}$ that define a Gaussian quadrature approximation such that any polynomial of degree $2m - 1$ or less is integrated exactly over $[a, b]$ with the given weight function.

[16] For a detailed overview of Gaussian quadrature, see J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 3rd ed. Springer, 2002.

[17] The weight function is often a probability density function in practice.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Given a domain and a weight function, we can compute a class of orthogonal polynomials. We will use $b_i(x)$ to denote an orthogonal polynomial[18] of degree i. Any polynomial of degree m can be represented as a linear combination of the orthogonal polynomials up to degree m. We form a quadrature rule by selecting points $\mathbf{x}_{1:m}$ to be the zeros of the orthogonal polynomial b_m and obtain the weights by solving the system of equations:

[18] Orthogonal polynomials are covered in chapter 21.

$$\sum_{i=1}^{m} b_k(x_i) w_i = \begin{cases} \int_a^b p(x) b_0(x)^2 \, dx & \text{for } k = 0 \\ 0 & \text{for } k = 1, \ldots, m-1 \end{cases} \tag{C.45}$$

Example C.4. Obtaining a 3-term quadrature rule for exactly integrating polynomials up to degree 5.

Consider the Legendre polynomials for integration over $[-1, 1]$ with the weight function $p(x) = 1$. Suppose our function of interest is well approximated by a 5th-degree polynomial. We construct a 3-point quadrature rule, which produces exact results for polynomials up to degree 5.

The Legendre polynomial of degree 3 is $\text{Le}_3(x) = \frac{5}{2}x^3 - \frac{3}{2}x$, which has roots at $x_1 = -\sqrt{3/5}$, $x_2 = 0$, and $x_3 = \sqrt{3/5}$. The Legendre polynomials of lesser degree are $\text{Le}_0(x) = 1$, $\text{Le}_1(x) = x$, and $\text{Le}_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$. The weights are obtained by solving the system of equations:

$$\begin{bmatrix} \text{Le}_0(-\sqrt{3/5}) & \text{Le}_0(0) & \text{Le}_0(\sqrt{3/5}) \\ \text{Le}_1(-\sqrt{3/5}) & \text{Le}_1(0) & \text{Le}_1(\sqrt{3/5}) \\ \text{Le}_2(-\sqrt{3/5}) & \text{Le}_2(0) & \text{Le}_2(\sqrt{3/5}) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} \text{Le}_0(x)^2 \, dx \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ -\sqrt{3/5} & 0 & \sqrt{3/5} \\ 4/10 & -1/2 & 4/10 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

which yields $w_1 = w_3 = 5/9$ and $w_2 = 8/9$.

Consider integrating the 5th-degree polynomial $f(x) = x^5 - 2x^4 + 3x^3 + 5x^2 - x + 4$. The exact value is $\int_{-1}^{1} p(x) f(x) \, dx = 158/15 \approx 10.533$. The quadrature rule produces the same value:

$$\sum_{i=1}^{3} w_i f(x_i) = \frac{5}{9} f\left(-\sqrt{\frac{3}{5}}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\sqrt{\frac{3}{5}}\right) \approx 10.533.$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Gauss solved equation (C.45) for the interval $[-1, 1]$ and the weighting function $p(x) = 1$. The orthogonal polynomials for this case are the *Legendre polynomials*. Algorithm C.1 implements Gaussian quadrature for Legendre polynomials and example C.4 works out a quadrature rule for integration over $[-1, 1]$.

We can transform any integral over the bounded interval $[a, b]$ to an integral over $[-1, 1]$ using the transformation

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)dx \tag{C.46}$$

Quadrature rules can thus be precalculated for the Legendre polynomials and then applied to integration over any finite interval.[19] Example C.5 applies such a transformation and algorithm C.2 implements integral transformations in a Gaussian quadrature method for finite domains.

[19] Similar techniques can be applied to integration over infinite intervals, such as $[0, \infty)$ using the Laguerre polynomials and $(-\infty, \infty)$ using the Hermite polynomials.

Example C.5. Integrals over finite regions can be transformed into integrals over $[-1, 1]$ and solved with quadrature rules for Legendre polynomials.

Consider integrating $f(x) = x^5 - 2x^4 + 3x^3 + 5x^2 - x + 4$ over $[-3, 5]$. We can transform this into an integration over $[-1, 1]$ using equation (C.46):

$$\int_{-3}^{5} f(x)\,dx = \frac{5+3}{2}\int_{-1}^{1} f\left(\frac{5+3}{2}x + \frac{5-3}{2}\right)dx = 4\int_{-1}^{1} f(4x+1)\,dx$$

We use the 3-term quadrature rule obtained in example C.4 to integrate $g(x) = 4f(4x+1) = 4096x^5 + 3072x^4 + 1280x^3 + 768x^2 + 240x + 40$ over $[-1, 1]$:

$$\int_{-1}^{1} p(x)g(x)\,dx = \frac{5}{9}g\left(-\sqrt{3/5}\right) + \frac{8}{9}g(0) + \frac{5}{9}g\left(\sqrt{3/5}\right) = 1820.8$$

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

```
struct Quadrule
    ws # weights
    xs # nodes
end
function quadrule_legendre(m)
    bs = [legendre(i) for i in 1 : m+1]
    xs = roots(bs[end])
    A = [bs[k](xs[i]) for k in 1 : m, i in 1 : m]
    b = zeros(m)
    b[1] = 2
    ws = A\b
    return Quadrule(ws, xs)
end
```

Algorithm C.1. A method for constructing `m`-point Legendre quadrature rules over $[-1, 1]$. The resulting type contains both the nodes `xs` and the weights `ws`.

```
quadint(f, quadrule) =
    sum(w*f(x) for (w,x) in zip(quadrule.ws, quadrule.xs))
function quadint(f, quadrule, a, b)
    α = (b-a)/2
    β = (a+b)/2
    g = x -> α*f(α*x+β)
    return quadint(g, quadrule)
end
```

Algorithm C.2. The function `quadint` for integrating a univariate function `f` with a given quadrature rule `quadrule` over the finite domain $[a, b]$.

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

References

1. A. Ahmadi-Javid, "Entropic Value-At-Risk: A New Coherent Risk Measure," *Journal of Optimization Theory and Applications*, vol. 155, no. 3, pp. 1105–1123, 2011 (cit. on p. 436).
2. N. M. Alexandrov and M. Y. Hussaini, eds., *Multidisciplinary Design Optimization: State of the Art*. SIAM, 1997 (cit. on p. 515).
3. S. Amari, "Natural Gradient Works Efficiently in Learning," *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998 (cit. on p. 97).
4. Aristotle, *Metaphysics*, trans. by W. D. Ross. 350 BCE, Book I, Part 5 (cit. on p. 2).
5. L. Armijo, "Minimization of Functions Having Lipschitz Continuous First Partial Derivatives," *Pacific Journal of Mathematics*, vol. 16, no. 1, pp. 1–3, 1966 (cit. on p. 65).
6. J. Arora, *Introduction to Optimum Design*, 4th ed. Academic Press, 2016 (cit. on p. 4).
7. R. K. Arora, *Optimization: Algorithms and Applications*. Chapman and Hall/CRC, 2015 (cit. on p. 6).
8. T. W. Athan and P. Y. Papalambros, "A Note on Weighted Criteria Methods for Compromise Solutions in Multi-Objective Optimization," *Engineering Optimization*, vol. 27, no. 2, pp. 155–176, 1996 (cit. on p. 334).
9. C. Audet and J. E. Dennis Jr., "Mesh Adaptive Direct Search Algorithms for Constrained Optimization," *SIAM Journal on Optimization*, vol. 17, no. 1, pp. 188–217, 2006 (cit. on pp. 117, 136).
10. D. A. Bader, W. E. Hart, and C. A. Phillips, "Parallel Algorithm Design for Branch and Bound," in *Tutorials on Emerging Methodologies and Applications in Operations Research*, H. J. Greenberg, ed., Kluwer Academic Press, 2004 (cit. on p. 471).
11. W. W. R. Ball, *Mathematical Recreations and Essays*. Macmillan, 1892 (cit. on p. 513).
12. D. Barber, *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012 (cit. on p. 502).

13. A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online Learning Rate Adaptation with Hypergradient Descent," in *International Conference on Learning Representations (ICLR)*, 2018 (cit. on p. 86).

14. A. D. Belegundu and T. R. Chandrupatla, *Optimization Concepts and Applications in Engineering*, 2nd ed. Cambridge University Press, 2011 (cit. on pp. 6, 314).

15. R. Bellman, "On the Theory of Dynamic Programming," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 38, no. 8, pp. 716–719, 1952 (cit. on p. 3).

16. R. Bellman, *Eye of the Hurricane: An Autobiography*. World Scientific, 1984 (cit. on p. 474).

17. H. Benaroya and S. M. Han, *Probability Models in Engineering and Science*. Taylor & Francis, 2005 (cit. on p. 444).

18. F. Berkenkamp, A. P. Schoellig, and A. Krause, "Safe Controller Optimization for Quadrotors with Gaussian Processes," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2016 (cit. on p. 417).

19. D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996 (cit. on p. 197).

20. M. Besançon, T. Papamarkou, D. Anthoff, A. Arslan, S. Byrne, D. Lin, and J. Pearson, "Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem," *Journal of Statistical Software*, vol. 98, no. 16, pp. 1–30, 2021 (cit. on p. 164).

21. H.-G. Beyer and B. Sendhoff, "Robust Optimization—A Comprehensive Survey," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 33, pp. 3190–3218, 2007 (cit. on p. 427).

22. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*. Springer, 2024 (cit. on p. 4).

23. T. L. Booth and R. A. Thompson, "Applying Probability Measures to Abstract Languages," *IEEE Transactions on Computers*, vol. C-22, no. 5, pp. 442–450, 1973 (cit. on p. 501).

24. C. Boutilier, R. Patrascu, P. Poupart, and D. Schuurmans, "Constraint-Based Optimization and Utility Elicitation Using the Minimax Decision Criterion," *Artificial Intelligence*, vol. 170, no. 8–9, pp. 686–713, 2006 (cit. on p. 343).

25. G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, 2nd ed. Wiley, 2005 (cit. on pp. 349, 427).

26. S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A Tutorial on Geometric Programming," *Optimization and Engineering*, vol. 8, pp. 67–127, 2007 (cit. on p. 247).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

27. S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011 (cit. on pp. 219, 236, 241).

28. S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004 (cit. on pp. 6, 77, 185, 209, 210, 576).

29. S. Boyd and L. Vandenberghe, *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018 (cit. on p. 276).

30. D. Braziunas and C. Boutilier, "Minimax Regret-Based Elicitation of Generalized Additive Utilities," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007 (cit. on p. 343).

31. D. Braziunas and C. Boutilier, "Elicitation of Factored Utilities," *AI Magazine*, vol. 29, no. 4, pp. 79–92, 2009 (cit. on p. 343).

32. L. Bregman, "The Relaxation Method of Finding the Common Point of Convex Sets and Its Application to the Solution of Problems in Convex Programming," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 3, pp. 200–217, 1967 (cit. on p. 231).

33. R. P. Brent, *Algorithms for Minimization Without Derivatives*. Prentice Hall, 1973 (cit. on p. 59).

34. M. D. Buhmann, *Radial Basis Functions*. Cambridge University Press, 2003 (cit. on p. 371).

35. F. Chaubard and M. J. Kochenderfer, "Scaling Recurrent Neural Networks to a Billion Parameters with Zero-Order Optimization," 2025. arXiv: 2505.17852 (cit. on p. 37).

36. F. H. Clarke, "Generalized Gradients and Applications," *Transactions of the American Mathematical Society*, vol. 205, pp. 247–262, 1975 (cit. on p. 577).

37. S. J. Colley, *Vector Calculus*, 4th ed. Pearson, 2011 (cit. on p. 23).

38. V. Conitzer, "Eliciting Single-Peaked Preferences Using Comparison Queries," *Journal of Artificial Intelligence Research*, vol. 35, pp. 161–191, 2009 (cit. on p. 340).

39. S. Cook, "The Complexity of Theorem-Proving Procedures," in *ACM Symposium on Theory of Computing*, 1971 (cit. on p. 481).

40. W. Cook, A. M. Gerards, A. Schrijver, and É. Tardos, "Sensitivity Theorems in Integer Linear Programming," *Mathematical Programming*, vol. 34, no. 3, pp. 251–264, 1986 (cit. on p. 465).

41. A. Corana, M. Marchesi, C. Martini, and S. Ridella, "Minimizing Multimodal Functions of Continuous Variables with the 'Simulated Annealing' Algorithm," *ACM Transactions on Mathematical Software*, vol. 13, no. 3, pp. 262–280, 1987 (cit. on p. 143).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

42. G. B. Dantzig, "Origins of the Simplex Method," in *A History of Scientific Computing*, S. G. Nash, ed., ACM, 1990, pp. 141–151 (cit. on p. 255).

43. S. Das and P. N. Suganthan, "Differential Evolution: A Survey of the State-of-the-Art," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011 (cit. on p. 172).

44. W. C. Davidon, "Variable Metric Method for Minimization," Argonne National Laboratory, Tech. Rep. ANL-5990, 1959 (cit. on p. 103).

45. W. C. Davidon, "Variable Metric Method for Minimization," *SIAM Journal on Optimization*, vol. 1, no. 1, pp. 1–17, 1991 (cit. on p. 103).

46. A. Dean, D. Voss, and D. Draguljić, *Design and Analysis of Experiments*, 2nd ed. Springer, 2017 (cit. on p. 349).

47. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002 (cit. on p. 336).

48. T. J. Dekker, "Finding a Zero by Means of Successive Linear Interpolation," in *Constructive Aspects of the Fundamental Theorem of Algebra*, B. Dejon and P. Henrici, eds., Interscience, 1969 (cit. on p. 59).

49. R. Descartes, "La Géométrie," in *Discours de la Méthode*. 1637 (cit. on p. 2).

50. S. Diamond and S. Boyd, "CVXPY: A Python-Embedded Modeling Language for Convex Optimization," *Journal of Machine Learning Research*, vol. 17, pp. 1–5, 2016 (cit. on p. 297).

51. E. D. Dolan, R. M. Lewis, and V. Torczon, "On the Local Convergence of Pattern Search," *SIAM Journal on Optimization*, vol. 14, no. 2, pp. 567–583, 2003 (cit. on p. 117).

52. A. Domahidi, E. Chu, and S. Boyd, "ECOS: An SOCP Solver for Embedded Systems," in *European Control Conference (ECC)*, 2013 (cit. on p. 297).

53. M. Dorigo, G. Di Caro, and L. M. Gambardella, "Ant Algorithms for Discrete Optimization," *Artificial Life*, vol. 5, no. 2, pp. 137–172, 1999 (cit. on p. 478).

54. M. Dorigo, V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996 (cit. on pp. 477, 478).

55. J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011 (cit. on p. 82).

56. R. L. Dykstra, "An Algorithm for Restricted Least Squares Regression," *Journal of the American Statistical Association*, vol. 78, no. 384, pp. 837–842, 1983 (cit. on p. 227).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

57. B. Efron, "Bootstrap Methods: Another Look at the Jackknife," *The Annals of Statistics*, vol. 7, pp. 1–26, 1979 (cit. on p. 379).

58. B. Efron, "Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation," *Journal of the American Statistical Association*, vol. 78, no. 382, pp. 316–331, 1983 (cit. on p. 381).

59. B. Efron and R. Tibshirani, "Improvements on Cross-Validation: The .632+ Bootstrap Method," *Journal of the American Statistical Association*, vol. 92, no. 438, pp. 548–560, 1997 (cit. on p. 381).

60. Euclid, *The Elements*, trans. by D. E. Joyce. 300 BCE (cit. on p. 2).

61. J. Fairbrother, C. Nemeth, M. Rischard, J. Brea, and T. Pinder, "GaussianProcesses.jl: A Nonparametric Bayes Package for the Julia Language," *Journal of Statistical Software*, vol. 102, no. 1, 2022 (cit. on p. 391).

62. R. Fletcher, *Practical Methods of Optimization*, 2nd ed. Wiley, 1987 (cit. on p. 103).

63. R. Fletcher and M. J. D. Powell, "A Rapidly Convergent Descent Method for Minimization," *The Computer Journal*, vol. 6, no. 2, pp. 163–168, 1963 (cit. on p. 103).

64. R. Fletcher and C. M. Reeves, "Function Minimization by Conjugate Gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 1964 (cit. on p. 80).

65. J. J. Forrest and D. Goldfarb, "Steepest-Edge Simplex Algorithms for Linear Programming," *Mathematical Programming*, vol. 57, no. 1, pp. 341–374, 1992 (cit. on p. 261).

66. A. Forrester, A. Sobester, and A. Keane, *Engineering Design via Surrogate Modelling: A Practical Guide*. Wiley, 2008 (cit. on p. 365).

67. A. I. J. Forrester, A. Sóbester, and A. J. Keane, "Multi-Fidelity Optimization via Surrogate Modelling," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 463, no. 2088, pp. 3251–3269, 2007 (cit. on p. 356).

68. J. H. Friedman, "Exploratory Projection Pursuit," *Journal of the American Statistical Association*, vol. 82, no. 397, pp. 249–266, 1987 (cit. on p. 445).

69. A. Fu, B. Ungun, L. Xing, and S. Boyd, "A Convex Optimization Approach to Radiation Treatment Planning with Dose Constraints," *Optimization and Engineering*, vol. 20, pp. 277–300, 2019 (cit. on p. 9).

70. R. Garnett, *Bayesian Optimization*. Cambridge University Press, 2023 (cit. on p. 411).

71. W. Gautschi, *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, 2004 (cit. on p. 450).

72. R. Ghanem, D. Higdon, and H. Owhadi, eds., *Handbook of Uncertainty Quantification*. Springer, 2017 (cit. on p. 443).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

73. A. Girard, C. E. Rasmussen, J. Q. Candela, and R. Murray-Smith, "Gaussian Process Priors with Uncertain Inputs—Application to Multiple-Step Ahead Time Series Forecasting," in *Advances in Neural Information Processing Systems* (*NIPS*), 2003 (cit. on p. 456).

74. D. E. Goldberg and J. Richardson, "Genetic Algorithms with Sharing for Multimodal Function Optimization," in *International Conference on Genetic Algorithms*, 1987 (cit. on p. 339).

75. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989 (cit. on pp. 164, 568).

76. T. Goldstein and S. Osher, "The Split Bregman Method for L_1-Regularized Problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 2, pp. 323–343, 2009 (cit. on p. 231).

77. G. H. Golub and J. H. Welsch, "Calculation of Gauss Quadrature Rules," *Mathematics of Computation*, vol. 23, no. 106, pp. 221–230, 1969 (cit. on p. 454).

78. R. E. Gomory, "An Algorithm for Integer Solutions to Linear Programs," *Recent Advances in Mathematical Programming*, vol. 64, pp. 269–302, 1963 (cit. on p. 466).

79. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016 (cit. on p. 7).

80. P. J. Goulart and Y. Chen, "Clarabel: An Interior-Point Solver for Conic Programs with Quadratic Objectives," 2024. arXiv: `2405.12762` (cit. on p. 297).

81. M. Grant and S. Boyd, "Graph Implementations for Nonsmooth Convex Programs," in *Recent Advances in Learning and Control*, 2008 (cit. on p. 311).

82. M. Grant, S. Boyd, and Y. Ye, "Disciplined Convex Programming," *Global Optimization: From Theory to Implementation*, pp. 155–210, 2006 (cit. on pp. 297, 315).

83. A. Grattafiori, A. Dubey, A. Jauhri, et al., "The Llama 3 Herd of Models," 2024. arXiv: `2407.21783` (cit. on p. 139).

84. A. Griewank, "On Automatic Differentiation," *Mathematical Programming: Recent Developments and Applications*, pp. 83–108, 1989 (cit. on p. 33).

85. A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008 (cit. on p. 26).

86. S. Gulwani, "Automating String Processing in Spreadsheets Using Input-Output Examples," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL*), 2011 (cit. on p. 489).

87. S. Guo and S. Sanner, "Real-Time Multiattribute Bayesian Preference Elicitation with Pairwise Comparison Queries," in *International Conference on Artificial Intelligence and Statistics* (*AISTATS*), 2010 (cit. on p. 340).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

88. B. Hajek, "Cooling Schedules for Optimal Annealing," *Mathematics of Operations Research*, vol. 13, no. 2, pp. 311–329, 1988 (cit. on p. 141).

89. T. C. Hales, "The Honeycomb Conjecture," *Discrete & Computational Geometry*, vol. 25, pp. 1–22, 2001 (cit. on p. 2).

90. J. H. Halton, "Algorithm 247: Radical-Inverse Quasi-Random Point Sequence," *Communications of the ACM*, vol. 7, no. 12, pp. 701–702, 1964 (cit. on p. 360).

91. N. Hansen, "The CMA Evolution Strategy: A Tutorial," 2016. arXiv: 1604.00772 (cit. on p. 152).

92. T. Hastie, R. Tibshirani, and J. H. Friedman, *The Elements of Statistical Learning*, 2nd ed. Springer, 2017 (cit. on p. 62).

93. B. S. He, H. Yang, and S. L. Wang, "Alternating Direction Method with Self-Adaptive Penalty Parameters for Monotone Variational Inequalities," *Journal of Optimization Theory and Applications*, vol. 106, no. 2, pp. 337–156, 2000 (cit. on p. 222).

94. F. M. Hemez and Y. Ben-Haim, "Info-Gap Robustness for the Correlation of Tests and Simulations of a Non-Linear Transient," *Mechanical Systems and Signal Processing*, vol. 18, no. 6, pp. 1443–1467, 2004 (cit. on p. 432).

95. M. R. Hestenes, "Multiplier and Gradient Methods," *Journal of Optimization Theory and Applications*, vol. 4, no. 5, pp. 303–320, 1969 (cit. on p. 197).

96. G. Hinton and S. Roweis, "Stochastic Neighbor Embedding," in *Advances in Neural Information Processing Systems (NIPS)*, 2003 (cit. on p. 135).

97. J.-B. Hiriart-Urruty and C. Lemaréchal, *Fundamentals of Convex Analysis*. Springer, 2001 (cit. on p. 579).

98. R. Hooke and T. A. Jeeves, "Direct Search Solution of Numerical and Statistical Problems," *Journal of the ACM (JACM)*, vol. 8, no. 2, pp. 212–229, 1961 (cit. on p. 116).

99. P. J. Huber, "Robust Estimation of a Location Parameter," *Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964 (cit. on p. 230).

100. H. Ishibuchi and T. Murata, "A Multi-Objective Genetic Local Search Algorithm and Its Application to Flowshop Scheduling," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 28, no. 3, pp. 392–403, 1998 (cit. on p. 336).

101. V. S. Iyengar, J. Lee, and M. Campbell, "Q-EVAL: Evaluating Multiple Attribute Items Using Queries," in *ACM Conference on Electronic Commerce*, 2001 (cit. on p. 342).

102. P. Jiang, Q. Zhou, and X. Shao, *Surrogate Model-Based Engineering Design and Optimization*. Springer, 2020 (cit. on p. 365).

103. D. Jones and M. Tamiz, *Practical Goal Programming*. Springer, 2010 (cit. on p. 332).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

104. D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian Optimization Without the Lipschitz Constant," *Journal of Optimization Theory and Application*, vol. 79, no. 1, pp. 157–181, 1993 (cit. on p. 124).

105. A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962 (cit. on p. 518).

106. L. Kallmeyer, *Parsing Beyond Context-Free Grammars*. Springer, 2010 (cit. on p. 489).

107. L. V. Kantorovich, "A New Method of Solving Some Classes of Extremal Problems," *Proceedings of the USSR Academy of Sciences*, vol. 28, pp. 211–214, 1940 (cit. on p. 3).

108. A. F. Kaupe Jr., "Algorithm 178: Direct Search," *Communications of the ACM*, vol. 6, no. 6, pp. 313–314, 1963 (cit. on p. 117).

109. A. Keane and P. Nair, *Computational Approaches for Aerospace Design*. Wiley, 2005 (cit. on p. 6).

110. J. Kennedy, R. C. Eberhart, and Y. Shi, *Swarm Intelligence*. Morgan Kaufmann, 2001 (cit. on p. 172).

111. D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations (ICLR)*, 2015 (cit. on p. 86).

112. S. Kiranyaz, T. Ince, and M. Gabbouj, *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*. Springer, 2014, Section 2.1 (cit. on p. 2).

113. S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983 (cit. on p. 140).

114. T. H. Kjeldsen, "A Contextualized Historical Analysis of the Kuhn-Tucker Theorem in Nonlinear Programming: The Impact of World War II," *Historia Mathematica*, vol. 27, no. 4, pp. 331–361, 2000 (cit. on p. 194).

115. L. Kocis and W. J. Whiten, "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, vol. 23, no. 2, pp. 266–294, 1997 (cit. on p. 361).

116. P. J. Kolesar, "A Branch and Bound Algorithm for the Knapsack Problem," *Management Science*, vol. 13, no. 9, pp. 723–735, 1967 (cit. on p. 483).

117. B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 6th ed. Springer, 2018 (cit. on p. 463).

118. J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap, "Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 2, pp. 109–128, 1997 (cit. on p. 489).

119. J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992 (cit. on p. 493).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

120. K. W. C. Ku and M.-W. Mak, "Exploring the Effects of Lamarckian and Baldwinian Learning in Evolving Recurrent Neural Networks," in *IEEE Congress on Evolutionary Computation (CEC)*, 1997 (cit. on p. 179).

121. L. Kuipers and H. Niederreiter, *Uniform Distribution of Sequences*. Dover, 2012 (cit. on p. 353).

122. J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 112–147, 1998 (cit. on p. 119).

123. R. Lam, K. Willcox, and D. H. Wolpert, "Bayesian Optimization with a Finite Budget: An Approximate Dynamic Programming Approach," in *Advances in Neural Information Processing Systems (NIPS)*, 2016 (cit. on p. 411).

124. A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960 (cit. on p. 471).

125. C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Prentice-Hall, 1974 (cit. on pp. 273, 288).

126. C. Lemieux, *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer, 2009 (cit. on p. 359).

127. J. R. Lepird, M. P. Owen, and M. J. Kochenderfer, "Bayesian Preference Elicitation for Multiobjective Engineering Design Optimization," *Journal of Aerospace Information Systems*, vol. 12, no. 10, pp. 634–645, 2015 (cit. on p. 340).

128. K. Levenberg, "A Method for the Solution of Certain Non-Linear Problems in Least Squares," *Quarterly of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, 1944 (cit. on pp. 71, 100).

129. Z. Lin, H. Li, and C. Fang, *Alternating Direction Method of Multipliers for Machine Learning*. Springer, 2022 (cit. on p. 219).

130. S. Linnainmaa, "The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors," M.S. thesis, University of Helsinki, 1970 (cit. on p. 33).

131. T. Lyche, *Numerical Linear Algebra and Matrix Factorizations*. Springer, 2020 (cit. on p. 582).

132. S. Malladi, T. Gao, E. Nichani, A. Damian, J. Lee, D. Chen, and S. Arora, "Fine-Tuning Language Models with Just Forward Passes," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023 (cit. on p. 139).

133. M. Manfrin, "Ant Colony Optimization for the Vehicle Routing Problem," Ph.D. dissertation, Université Libre de Bruxelles, 2004 (cit. on p. 477).

134. H. Markowitz, "Portfolio Selection," *Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952 (cit. on p. 438).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

135. R. T. Marler and J. S. Arora, "Survey of Multi-Objective Optimization Methods for Engineering," *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, 2004 (cit. on p. 325).

136. D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963 (cit. on p. 100).

137. J. Martens, *Journal of Machine Learning Research*, vol. 21, pp. 1–76, 2020 (cit. on p. 149).

138. J. R. R. A. Martins and A. B. Lambe, "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, vol. 51, no. 9, pp. 2049–2075, 2013 (cit. on p. 515).

139. J. R. R. A. Martins and A. Ning, *Engineering Design Optimization*. Cambridge University Press, 2022 (cit. on p. 515).

140. J. R. R. A. Martins, P. Sturdza, and J. J. Alonso, "The Complex-Step Derivative Approximation," *ACM Transactions on Mathematical Software*, vol. 29, no. 3, pp. 245–262, 2003 (cit. on p. 28).

141. J. H. Mathews and K. D. Fink, *Numerical Methods Using MATLAB*, 4th ed. Pearson, 2004 (cit. on p. 27).

142. K. Miettinen, *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, 1999 (cit. on p. 325).

143. J. E. Mitchell, "Branch-And-Cut Algorithms for Combinatorial Optimization Problems," in *Handbook of Applied Optimization*, P. M. Pardalos and M. G. C. Resende, eds., Oxford University Press, 2002, pp. 65–77 (cit. on p. 471).

144. D. J. Montana, "Strongly Typed Genetic Programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995 (cit. on p. 493).

145. D. C. Montgomery, *Design and Analysis of Experiments*. Wiley, 2017 (cit. on p. 349).

146. M. D. Morris and T. J. Mitchell, "Exploratory Designs for Computational Experiments," *Journal of Statistical Planning and Inference*, vol. 43, no. 3, pp. 381–402, 1995 (cit. on p. 355).

147. K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012 (cit. on pp. 366, 375).

148. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022 (cit. on p. 8).

149. S. Narayanan and S. Azarm, "On Improving Multiobjective Genetic Algorithms for Design Optimization," *Structural Optimization*, vol. 18, no. 2–3, pp. 146–155, 1999 (cit. on p. 339).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

150. S. Nash and A. Sofer, *Linear and Nonlinear Programming*. McGraw-Hill, 1996 (cit. on p. 210).

151. J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965 (cit. on p. 119).

152. A. S. Nemirovski and M. J. Todd, "Interior-Point Methods for Optimization," *Acta Numerica*, vol. 17, pp. 191–234, 2008 (cit. on p. 199).

153. Y. Nesterov, "A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$," *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 543–547, 1983 (cit. on p. 82).

154. J. Nocedal, "Updating Quasi-Newton Matrices with Limited Storage," *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980 (cit. on p. 105).

155. J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. Springer, 2006 (cit. on pp. 63, 68).

156. J. Nocedal and S. J. Wright, "Trust-Region Methods," in *Numerical Optimization*, T. V. Mikosch, S. I. Resnick, and S. M. Robinson, eds. Springer, 2006, pp. 66–100 (cit. on p. 73).

157. B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd, "Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding," *Journal of Optimization Theory and Applications*, vol. 169, no. 3, pp. 1042–1068, 2016 (cit. on p. 298).

158. A. O'Hagan, "Some Bayesian Numerical Analysis," in *Bayesian Statistics*, J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, eds., vol. 4, Oxford University Press, 1992, pp. 345–363 (cit. on p. 396).

159. Y. Ollivier, L. Arnold, A. Auger, and N. Hansen, "Information-Geometric Optimization Algorithms: A Unifying Picture via Invariance Principles," *Journal of Machine Learning Research*, vol. 18, no. 18, pp. 1–65, 2017 (cit. on p. 149).

160. M. Padberg and G. Rinaldi, "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems," *SIAM Review*, vol. 33, no. 1, pp. 60–100, 1991 (cit. on p. 466).

161. P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design*. Cambridge University Press, 2017 (cit. on p. 6).

162. N. Parikh and S. Boyd, "Proximal Algorithms," *Foundations and Trends in Optimization*, vol. 1, no. 3, pp. 127–239, 2014 (cit. on p. 224).

163. G.-J. Park, T.-H. Lee, K. H. Lee, and K.-H. Hwang, "Robust Design: An Overview," *AIAA Journal*, vol. 44, no. 1, pp. 181–191, 2006 (cit. on p. 427).

164. S. K. Park, "A Transformation Method for Constrained-Function Minimization," National Aeronautics and Space Administration, Technical Note TN D-7983, 1975 (cit. on p. 185).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

165. J. Peters and S. Schaal, "Reinforcement Learning of Motor Skills with Policy Gradients," *Neural Networks*, vol. 21, no. 4, pp. 682–697, 2008 (cit. on p. 35).

166. G. C. Pflug, "Some Remarks on the Value-at-Risk and the Conditional Value-at-Risk," in *Probabilistic Constrained Optimization: Methodology and Applications*, S. P. Uryasev, ed. Springer, 2000, pp. 272–281 (cit. on p. 439).

167. S. Piyavskii, "An Algorithm for Finding the Absolute Extremum of a Function," *USSR Computational Mathematics and Mathematical Physics*, vol. 12, no. 4, pp. 57–67, 1972 (cit. on p. 53).

168. E. Polak and G. Ribière, "Note sur la Convergence de Méthodes de Directions Conjuguées," *Revue Française d'informatique et de Recherche Opérationnelle, Série Rouge*, vol. 3, no. 1, pp. 35–43, 1969 (cit. on p. 81).

169. B. T. Polyak, "Some Methods of Speeding up the Convergence of Iteration Methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964 (cit. on p. 91).

170. M. J. D. Powell, "An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives," *Computer Journal*, vol. 7, no. 2, pp. 155–162, 1964 (cit. on p. 115).

171. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1982 (cit. on p. 115).

172. K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization*. Springer, 2006 (cit. on p. 172).

173. P. Raghavan and C. D. Tompson, "Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987 (cit. on p. 465).

174. C. E. Rasmussen and Z. Ghahramani, "Bayesian Monte Carlo," in *Advances in Neural Information Processing Systems (NIPS)*, 2003 (cit. on p. 456).

175. C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. MIT Press, 2006 (cit. on pp. 391, 392, 401).

176. A. Ratle and M. Sebag, "Genetic Programming and Domain Knowledge: Beyond the Limitations of Grammar-Guided Machine Discovery," in *International Conference on Parallel Problem Solving from Nature*, 2000 (cit. on p. 512).

177. R. G. Regis, "On the Properties of Positive Spanning Sets and Positive Bases," *Optimization and Engineering*, vol. 17, no. 1, pp. 229–262, 2016 (cit. on p. 118).

178. A. M. Reynolds and M. A. Frye, "Free-Flight Odor Tracking in Drosophila is Consistent with an Optimal Intermittent Scale-Free Search," *PLoS ONE*, vol. 2, no. 4, e354, 2007 (cit. on p. 177).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

179. R. T. Rockafellar and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, vol. 2, pp. 21–42, 2000 (cit. on p. 436).

180. R. T. Rockafellar and S. Uryasev, "Conditional Value-at-Risk for General Loss Distributions," *Journal of Banking and Finance*, vol. 26, pp. 1443–1471, 2002 (cit. on p. 439).

181. H. H. Rosenbrock, "An Automatic Method for Finding the Greatest or Least Value of a Function," *The Computer Journal*, vol. 3, no. 3, pp. 175–184, 1960 (cit. on p. 567).

182. R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Springer, 2004 (cit. on p. 144).

183. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533–536, 1986 (cit. on p. 33).

184. C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *European Conference on Genetic Programming*, 1998 (cit. on p. 497).

185. E. K. Ryu and W. Yin, *Large-Scale Convex Optimization: Algorithms and Analyses via Monotone Operators*. Cambridge University Press, 2023 (cit. on p. 219).

186. T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864 (cit. on p. 148).

187. R. Salustowicz and J. Schmidhuber, "Probabilistic Incremental Program Evolution," *Evolutionary Computation*, vol. 5, no. 2, pp. 123–141, 1997 (cit. on p. 502).

188. J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," in *International Conference on Genetic Algorithms and Their Applications*, 1985 (cit. on p. 334).

189. C. Schretter, L. Kobbelt, and P.-O. Dehaye, "Golden Ratio Sequences for Low-Discrepancy Sampling," *Journal of Graphics Tools*, vol. 16, no. 2, pp. 95–104, 2016 (cit. on p. 359).

190. A. Shapiro, "Distributionally Robust Stochastic Programming," *SIAM Journal on Optimization*, vol. 27, no. 4, pp. 2258–2275, 2017 (cit. on p. 434).

191. A. Shapiro, D. Dentcheva, and A. Ruszczyński, *Lectures on Stochastic Programming: Modeling and Theory*, 2nd ed. SIAM, 2014 (cit. on p. 434).

192. A. Shmygelska, R. Aguirre-Hernández, and H. H. Hoos, "An Ant Colony Algorithm for the 2D HP Protein Folding Problem," in *International Workshop on Ant Algorithms (ANTS)*, 2002 (cit. on p. 477).

193. B. O. Shubert, "A Sequential Method Seeking the Global Maximum of a Function," *SIAM Journal on Numerical Analysis*, vol. 9, no. 3, pp. 379–388, 1972 (cit. on p. 53).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

194. D. Simon, *Evolutionary Optimization Algorithms*. Wiley, 2013 (cit. on p. 179).

195. J. Sobieszczanski-Sobieski, A. Morris, and M. van Tooren, *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*. Wiley, 2015 (cit. on p. 515).

196. I. M. Sobol, "On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, 1967 (cit. on p. 361).

197. D. C. Sorensen, "Newton's Method with a Model Trust Region Modification," *SIAM Journal on Numerical Analysis*, vol. 19, no. 2, pp. 409–426, 1982 (cit. on p. 71).

198. K. Sörensen, "Metaheuristics—the Metaphor Exposed," *International Transactions in Operational Research*, vol. 22, no. 1, pp. 3–18, 2015 (cit. on p. 179).

199. J. C. Spall, "Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation," *IEEE Transactions on Automatic Control*, vol. 37, pp. 332–341, 1992 (cit. on p. 37).

200. J. C. Spall, *Introduction to Stochastic Search and Optimization*. Wiley, 2003 (cit. on pp. 35, 37).

201. B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An Operator Splitting Solver for Quadratic Programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020 (cit. on p. 297).

202. T. J. Stieltjes, "Quelques Recherches sur la Théorie des Quadratures Dites Mécaniques," *Annales Scientifiques de l'École Normale Supérieure*, vol. 1, pp. 409–426, 1884 (cit. on p. 450).

203. J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 3rd ed. Springer, 2002 (cit. on pp. 97, 586).

204. M. Stone, "Cross-Validatory Choice and Assessment of Statistical Predictions," *Journal of the Royal Statistical Society*, vol. 36, no. 2, pp. 111–147, 1974 (cit. on p. 379).

205. T. Stützle, "MAX-MIN Ant System for Quadratic Assignment Problems," Technical University Darmstadt, Tech. Rep., 1997 (cit. on p. 477).

206. Y. Sui, A. Gotovos, J. Burdick, and A. Krause, "Safe Exploration for Optimization with Gaussian Processes," in *International Conference on Machine Learning (ICML)*, vol. 37, 2015 (cit. on p. 416).

207. G. Szegö, *Orthogonal Polynomials*, 4th ed. American Mathematical Society, 2012 (cit. on p. 448).

208. H. Szu and R. Hartley, "Fast Simulated Annealing," *Physics Letters A*, vol. 122, no. 3–4, pp. 157–162, 1987 (cit. on p. 141).

209. G. B. Thomas, *Calculus and Analytic Geometry*, 9th ed. Addison-Wesley, 1968 (cit. on p. 26).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

210. R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996 (cit. on p. 233).

211. V. Torczon, "On the Convergence of Pattern Search Algorithms," *SIAM Journal of Optimization*, vol. 7, no. 1, pp. 1–25, 1997 (cit. on p. 117).

212. M. Toussaint, "The Bayesian Search Game," in *Theory and Principled Methods for the Design of Metaheuristics*, Y. Borenstein and A. Moraglio, eds. Springer, 2014, pp. 129–144 (cit. on p. 411).

213. K. Truemper, "A Decomposition Theory for Matroids. V. Testing of Matrix Total Unimodularity," *Journal of Combinatorial Theory, Series B*, vol. 49, no. 2, pp. 241–281, 1990 (cit. on p. 467).

214. R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, 4th ed. Springer, 2014 (cit. on p. 249).

215. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2017 (cit. on p. 4).

216. J. Von Neumann, "On Rings of Operators. Reduction Theory," *Annals of Mathematics*, pp. 401–485, 1949 (cit. on p. 241).

217. D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, no. 15, pp. 949–980, 2014 (cit. on pp. 148–150).

218. H. P. Williams, *Model Building in Mathematical Programming*, 5th ed. Wiley, 2013 (cit. on p. 249).

219. P. Wolfe, "Convergence Conditions for Ascent Methods," *SIAM Review*, vol. 11, no. 2, pp. 226–235, 1969 (cit. on pp. 65, 68).

220. P. Wolfe, "Convergence Conditions for Ascent Methods. II: Some Corrections," *SIAM Review*, vol. 13, no. 2, pp. 185–188, 1971 (cit. on p. 65).

221. D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997 (cit. on p. 7).

222. P. K. Wong, L. Y. Lo, M. L. Wong, and K. S. Leung, "Grammar-Based Genetic Programming with Bayesian Network," in *IEEE Congress on Evolutionary Computation* (*CEC*), 2014 (cit. on p. 502).

223. S. J. Wright, *Primal-Dual Interior-Point Methods*. SIAM, 1997 (cit. on p. 213).

224. T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, 2023 (cit. on p. 4).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

225. H. Xiong, S. Shi, D. Ren, and J. Hu, "A Survey of Job Shop Scheduling Problem: The Types and Models," *Computers and Operations Research*, vol. 142, p. 105731, 2022 (cit. on p. 10).

226. X.-S. Yang, "A Brief History of Optimization," in *Engineering Optimization*. Wiley, 2010, pp. 3–10 (cit. on p. 2).

227. X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms*, 2nd ed. Luniver Press, 2010 (cit. on p. 174).

228. X.-S. Yang and S. Deb, "Cuckoo Search via Lévy Flights," in *World Congress on Nature & Biologically Inspired Computing (NaBIC)*, 2009 (cit. on p. 177).

229. P. L. Yu, "Cone Convexity, Cone Extreme Points, and Nondominated Solutions in Decision Problems with Multiobjectives," *Journal of Optimization Theory and Applications*, vol. 14, no. 3, pp. 319–377, 1974 (cit. on p. 332).

230. Y. X. Yuan, "Recent Advances in Trust Region Algorithms," *Mathematical Programming*, vol. 151, no. 1, pp. 249–281, 2015 (cit. on p. 71).

231. L. Zadeh, "Optimality and Non-Scalar-Valued Performance Criteria," *IEEE Transactions on Automatic Control*, vol. 8, no. 1, pp. 59–60, 1963 (cit. on p. 331).

232. M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," 2012. arXiv: `1212.5701` (cit. on p. 84).

233. Q. Zhou, M. Zhao, J. Hu, and M. Ma, *Multi-Fidelity Surrogates: Modeling, Optimization and Applications*. Springer, 2023 (cit. on p. 384).

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

Index

0.632 bootstrap estimate, 381

abstract types, 553
Ackley's function, 561
active, 192
Adadelta, 84
AdaGrad, *see* adaptive gradient
Adam, *see* adaptive moment estimation
adaptive gradient, 82
adaptive moment estimation, 86
additive recurrence, 359
ADMM, *see* alternating direction method of multipliers
affine subspace, 253
aleatory uncertainty, *see* irreducible uncertainty
algebra, 2
algorithm, 2
algorithms, 1
algoritmi, 2
alternating direction method of multipliers, 219
alternating projections, 227
anchoring point, 116
anonymous function, 555
ant colony optimization, 477
approximate line search, 65
Armijo condition, *see* sufficient decrease
Armijo line search, *see* backtracking line search
array comprehension, 544
assignment, 516
associative array, 516
asymptotic notation, 571
atom library, 300
augmented Lagrange method, *see* method of multipliers
augmented system, 316
automatic differentiation, 29
auxiliary linear program, 262

backslash operator, 36, 274
backtracking line search, 65
backward difference, 24
backward pass, 33
Baldwinian learning, 179
barrier methods, 199
basic, 257
basis functions, 367
basis pursuit, 231
batches, 135
Bayes-Hermite Quadrature, 455
Bayesian Monte Carlo, 455
Bayesian optimization, 411
BFGS, *see* Broyden-Fletcher-Goldfarb-Shanno
big *M* method, 485
big-Oh notation, 571
Binet's formula, 46
bisection method, 57
black box, 113
Bland's rule, 262
block-diagonal, 316
Boolean, 541
Boolean satisfiability problem, 481
Booth's function, 561
bootstrap method, 379
bootstrap sample, 379
bound operation, 471
bowl-shaped, 575
bowl, 11f
box constraint, 183
bracket, 43
bracketing, 43
bracketing phase, 68
branch, 471
branch and bound, 471
branch and cut, 466
branch operation, 471
Branin function, 564
Brent-Dekker, 57
broadcasting, 546
Broyden-Fletcher-Goldfarb-Shanno, 103

calculus, 3

canonical form, 297
canonicalization, 307
Cauchy distribution, 164
ceiling, 471
central difference, 24
characteristic length-scale, 392
Chebyshev distance, 577
chessboard distance, 577
Cholesky decomposition, 582
chromosome, 165
circle function, 569
CMA-ES, *see* covariance matrix adaptation
coherent risk measure, 436
collaborative optimization, 531
combinatorial optimization, 463
complete cross-validation, 379
complex step method, 28
composite type, 553
computational graph, 30
concave, 576
concrete types, 553
conditional distribution, 390
conditional value at risk, 436
cone, 185
cone constraint, 185
confidence region, 396
conjugate gradient, 79
consensus, 234
constraint, 6
constraint method, 329
context-free grammars, 489
contour plot, 13
contours, 13
convex combination, 574
convex function, 575
convex program, 297
convex set, 575
coordinate descent, 77, 113
coprime, 361
corrector function, 407
coupling variables, 526
covariance function, 391
covariance matrix adaptation, 152
covariance matrix adaptation evolutionary strategy, *see* covariance matrix adaptation
covariant gradient, 97
criterion space, 327
critical point, 191
cross-entropy, 144
cross-entropy method, 144
crossover, 165
crossover point, 168
cuckoo search, 177
cumulative distribution function, 436, 586
curvature, 299
curvature condition, 68, 110
cutting plane, 467
cutting plane method, 466
CVaR, *see* conditional value at risk
cycles, 262
cyclic coordinate search, 113

Dantzig's rule, 262
Davidon-Fletcher-Powell, 103
DCP, *see* disciplined convex program
decision quality improvement, 343
deep learning, 7
Dempster-Shafer theory, 428
dependency cycle, 518
dependency graph, 518
dequeue, 471
derivative-free, 113
derivative, 23
descent direction, 61
descent direction methods, 61
designer, 4
design matrix, 366
design point, 5
design selection, 343
design variables, 5
DFP, *see* Davidon-Fletcher-Powell
dictionary, 516, 552
differential evolution, 172
DIRECT, *see* divided rectangles
directional derivative, Clarke, 577
directional derivative, 26
direct methods, 113
Dirichlet distribution, 502
disciplinary analyses, 515
disciplined convex program, 297
discrepancy, 353
discrete factors, 349
discrete optimization, 463
dispatch, 557
distributed architecture, 532
distributionally robust optimization, 434
divided rectangles, 124
dominates, 326
dual ascent, 218
dual certificates, 266
dual feasibility, 193
dual form, 210
dual function, 210
duality, 209
duality gap, 210
dual numbers, 32
dual part, 32
dual residual, 221
dual value, 210
dual variable, 193, 209
dynamic ordering, 119
dynamic programming, 474

elastic band, 243
elite samples, 144
elitism, 165
elitist selection, *see* elitism
enqueue, 471
entering index, 260

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

entropic value at risk, 436
epigraph, 313
epistemic uncertainty, 427
equality form, 252
equivalence class sharing, 339
error-based exploration, 412
error function, 586
Euclidean norm, 577
evolution path, 154
exchange subset selection, 357
expected improvement, 415
expected value, 434
expert responses, 342
exploitation, 413
exploration, 413
exponential annealing schedule, 141
exponential weighted criterion, 334
extended-valued function, 300
extended real-valued function, 300

factor of safety, 440
fast annealing, 141
feasibility problem, 299
feasible set, 5
Fibonacci search, 45
fidelity, 383
finite difference methods, 27
firefly algorithm, 174
first-order, 77
first-order necessary condition, 11
first fundamental theorem of calculus, 573
Fisher information matrix, 149
fitness proportionate selection, *see* roulette wheel selection
fitness sharing, 339
Fletcher-Reeves, 80
floor, 469
flower function, 565
forward accumulation, 31
forward difference, 24
forward pass, 33
Fourier series, 370
fractional knapsack, 483
full factorial, 349
function, 555
fuzzy-set theory, 428

gamma function, 392
gap, 432
Gauss-Seidel method, 518
Gaussian distribution, 389
Gaussian process, 389, 391
Gaussian quadrature, 586
gene, 164
gene duplication, 497
general consensus optimization, 235
general form, 251
generalization error, 375
generalized gradient, Clarke, 577
generalized gradient, 577
generalized inequality constraint, 185
generalized Lagrangian, 193, 209
generalized pattern search, 117
generation, 163
genetic algorithms, 164
genetic local search, 179
genetic programming, 493
geometric program, 247
global design variables, 524
global minimizer, 10
global optimization method, 53
goal programming, 332
golden ratio, 46
golden section search, 47
gradient, 25
gradient descent, 77
grammar, 489
grammatical evolution, 497
graph expansion, 314
graph implementation, 314
greedy heuristic, 261
greedy subset selection, 357
grid search, 349

half-space, 252
Halton sequence, 360
heavy ball, 91
Hessian, 25
hill, 12
holdout method, 377
Hooke-Jeeves method, 116
Huber function, 230
hybrid methods, 179
hypergradient, 86
hypergradient descent, 86
hyperparameter, 44
hyperplane, 25
hypograph, 313

image, 327
improvement, 413
inactive, 193
individual discipline feasible, 526
individuals, 163
information-gap decision theory, 432
information-geometric optimization, 149
initialization phase, 258, 262
initial population, 163
integer linear programs, 464
integer program, 464
interdisciplinary compatibility, 517
interior point methods, 199
intermediate value theorem, 57
interpolation crossover, 168
inverse barrier, 199
irreducible uncertainty, 427
iterates, 61

Kahn's algorithm, 518
kernel, 391
keyword argument, 556
k-fold cross-validation, 378

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

KKT conditions, 194
knapsack problem, 475
Kullback-Leibler divergence, 149
Kullback–Leibler divergence, 144

L-BFGS, *see* Limited-memory BFGS
L_2 regularization, 372
Lagrange multiplier, 191
Lagrange multipliers, 189
Lagrangian, 191
Lamarckian learning, 179
lasso, 233, 372
Latin-hypercube sampling, 351
Latin squares, 351
LDL decomposition, 583
leaped Halton method, 361
learning rate, *see* step factor
least-squares problem, 274
least-squares problem with linear inequality constraints, 278
least absolute deviation problem, 229
least distance program, 281
leave-one-out bootstrap estimate, 381
leave-one-out cross-validation, 379
leaving index, 260
Lebesgue measure, 353
Legendre polynomials, 588
Levenberg-Marquardt algorithm, 100
lexicographic method, 330
Limited-memory BFGS, 105
limit supremum, 577
linear combination, 571
linear model, 366
linear program, 249
linear regression, 35, 366
line search, 64
Lipschitz continuous, 53
local design variables, 524
local minimum, 10
local models, 61
logarithmic annealing schedule, 141
log barrier, 199
log likelihood, 401
loop, 558
low-discrepancy sequences, 359
lower confidence bound, 413
lower confidence bound exploration, 413
LQ decomposition, 584
Lévy flights, 177

marginal distribution, 390
Markowitz portfolio optimization, 438
matrix, 548
matrix decomposition, 582
matrix factorization, *see* matrix decomposition
matrix norm, 110
Matérn kernel, 392
max-min inequality, 210
maximum likelihood estimate, 401
max norm, 577
MDO, *see* multidisciplinary design optimization
mean, 434
mean excess loss, 436
mean function, 391
mean shortfall, 436
mean squared error, 375
memetic algorithms, 179
memoization, 475
memory-efficient zeroth-order optimizer, 139
mesh adaptive direct search, 136
method of multipliers, 197
Metropolis criterion, 141
MeZO, *see* memory-efficient zeroth-order optimizer
Michalewicz function, 566
minimax, 429
minimax decision, 343
minimax regret, 343
minimizer, 5
minimum ratio test, 260
mixed-integer program, 464
modified Bessel function of the second kind, 392
momentum, 81
monomial, 247
Monte Carlo integration, 359, 443
Morris-Mitchell criterion, 355
multidisciplinary analysis, 517
multidisciplinary design feasible, 521
multidisciplinary design optimization, 515
multifidelity surrogate model, 383
multimodal function, 43
multiobjective optimization, 325
multivariate, 61
multivariate function, 10
mutate, 545
mutation, 165
mutation rate, 170
mutually conjugate, 79

named function, 555
named tuple, 552
natural evolution strategies, 149
natural gradient, 97, 149
necessary, 11
Nelder-Mead simplex method, 119
Nesterov momentum, 82
neural network kernel, 392
Newton's method, 96
niche, 339
niche techniques, 339
no free lunch theorems, 7
non-basic, 257
nondomination ranking, 336
nonnegative least-squares quadratic program, 282
nonparametric, 402
nonterminal, 489

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

nonterminal symbols, 489
norm, 577
normal distribution, 389, 586
normal equation, 385
numerical differentiation, 26

objective function, 5
operation overloading, 35
opportunistic, 119
optimal substructure, 474
optimization phase, 258, 262
order, 571
orthogonal, 447
orthogonal matrix, 584
orthonormal matrix, *see* orthogonal matrix
outer product approximation, 102
outgoing neighbors, 478
over-constrained, 252
overfitting, 232, 375
overlapping subproblems, 474

package, 560
paired query selection, 342
pairwise distances, 353
parametric, 402
parametric types, 554
Pareto curve, 327
Pareto filter, 336
Pareto frontier, 327
Pareto optimality, 325, 327
parsimony, 493
partial derivative, 25
partially observable Markov decision process, 411
particle, 172
particle swarm optimization, 172
partitioned canonical form, 310
pattern, 117
pattern search, 113
penalty methods, 195
pheromone, 477
pivoting, 260
Polak-Ribière, 81
polyhedral method, 343
polynomial basis functions, 369
polynomial chaos, 445
polytopes, 256
population methods, 163
positive definite, 582
positive definiteness, 582
positive semidefinite, 582
positive spanning set, 117
possibility theory, 428
posterior distribution, 395
Powell's method, 115
prediction-based exploration, 411
preference elicitation, 340
primal-dual method, 213
primal, 193
primal form, 210
primal residual, 220
primal value, 210
prime analytic center, 342
priority queue, 471
probabilistic grammar, 501
probabilistic prototype tree, 502
probabilistic uncertainty, 432
probability of improvement, 414
product-free expressions, 301
product correlation rule, 456
production rules, 489
projected descent, 189
proposal distribution, 144
proposal distribution descent, 148
proximal minimization, 224
pruning, 500
pseudo-random, 135
pseudoinverse, 277, 367

Q-Eval, 342
QR decomposition, 584
quadratic convergence, 96
quadratic fit search, 51
quadratic penalties, 196
quadratic program, 273
quadrature rule, 586
quantile, 436
quasi-Monte Carlo methods, 359
quasi-Newton, 103
quasi-random sequences, 359

radial function, 371
randomized rounding, 465
random restarts, 74
random sampling, 350
random subsampling, 377
random uncertainty, *see* irreducible uncertainty
real part, 32
reducible uncertainty, *see* epistemic uncertainty
regression, 366
regret, 343
regularization, 232
regularization term, 372
relax, 465
residual form, 534
response variables, 515
restricted step method, 71
reverse accumulation, 31, 33
ridge regression, 372
risk measure, 436
RMSProp, 84
robust, 436
robust counterpart approach, *see* minimax
robust regularization, *see* minimax
root-finding methods, 57
root mean squared error, 383
roots, 57
Rosenbrock function, 567
rotation estimation, 378
roulette wheel selection, 165

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

rounding, 465

saddle, 12
safe exploration, 416
SafeOpt, 416
sample mean, 443
sample variance, 443
sampling plans, 18, 349
scaled dual variable, 223
scaled form, 222
secant equation, 110
secant method, 97
second-order, 95
second-order necessary condition, 11
second-order sufficient condition, 14
second order cone program, 311
self-dual simplex algorithm, 267
semidefinite programming, 185
sensitive, 436
sequential optimization, 524
set-based uncertainty, 429
Shubert-Piyavskii method, 53
sign, 301
simplex, 119
simplex algorithm, 255
simulated annealing, 140
simultaneous analysis and design, 534
simultaneous perturbation stochastic approximation, 35
simultaneous perturbation stochastic gradient approximation, 37
single-point crossover, 168
singular value decomposition, 276, 585
singular values, 585
sinusoidal basis functions, 370
six-sigma, 441
slack, 194
slack variable, 195
Slater's condition, 210
Sobol sequences, 361
soft thresholding operator, 230
solution, 5
space-filling, 349
space-filling metrics, 352
space-filling subsets, 356
specification, 4
splat, 557
split Bregman method, 231
SPSA, *see* simultaneous perturbation stochastic gradient approximation
square-root-free Cholesky decomposition, *see* LDL decomposition
squared exponential kernel, 392
standard deviation, 396
standard form, 251
standard normal cumulative distribution function, 414
start type, 489
stationary point, 11, 78
statistical feasibility, 436
steepest descent, 77
step factor, 62
step size, 62
Stieltjes algorithm, 450
stochastic gradient descent, 135
stochastic methods, 135
stratified sampling, 352
strict local minimizer, 10
strictly concave, 576
strictly convex, 575
string, 543
strong backtracking line search, 68
strong curvature condition, 68
strong duality, 210
strong local minima, 10
strong local minimizer, 10
strongly typed genetic programming, 493
strong Wolfe conditions, 68
subdifferential, Clarke, 579
subdifferential, 579
subgradient, 579
submatrix, 465
suboptimizer, 521
subpopulations, 334
subproblem, 524
sufficient closeness, 97
sufficient decrease, 65
supremum, 353
surrogate model, 349
surrogate models, 365
symbolic differentiation, 24
symbols, 489
system-level optimizer, 524

tail value at risk, 436
taxicab search, 113
Taylor approximation, 574
Taylor expansion, 573
Taylor series, 573
Temperature, 140
terminal, 489
terminal symbols, 489
ternary operator, 558
test functions, 561
test set, 377
tight, 194
topological ordering, 518
totally unimodular, 466
tournament selection, 165
trace, 401
training error, 375
training set, 377
traveling salesman problem, 477
tree crossover, 493
tree mutation, 493
tree permutation, 495
triangle inequality, 577
truncation selection, 165
trust region, 71
tuple, 551
two-point crossover, 168

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com

types, 489

uncertainty propagation, 443
uniform crossover, 168
uniform projection plan, 351
unimodal function, 43
unimodality, 43
univariate function, 10
univariate Gaussian, 586
utopia point, 327

value at risk, 436
van der Corput sequences, 360
VaR, *see* value at risk
variance, 434
vector, 544
vector evaluated genetic algorithm, 334
vector optimization, 325
verification, 298
vertices, 256

warm start, 61
weak duality, 210
weak local minima, 10
weakly Pareto-optimal, 327
weighted exponential sum, 332
weighted min-max method, 333
weighted sum method, 331
weighted Tchebycheff method, 333
Wheeler's ridge, 568
whitening, 445
Wolfe conditions, 65

zero-order, 113
zero-order stochastic step, 139
zoom phase, 70

© 2026 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2026-09-05 15:07:21-07:00, comments to bugs@algorithmsbook.com