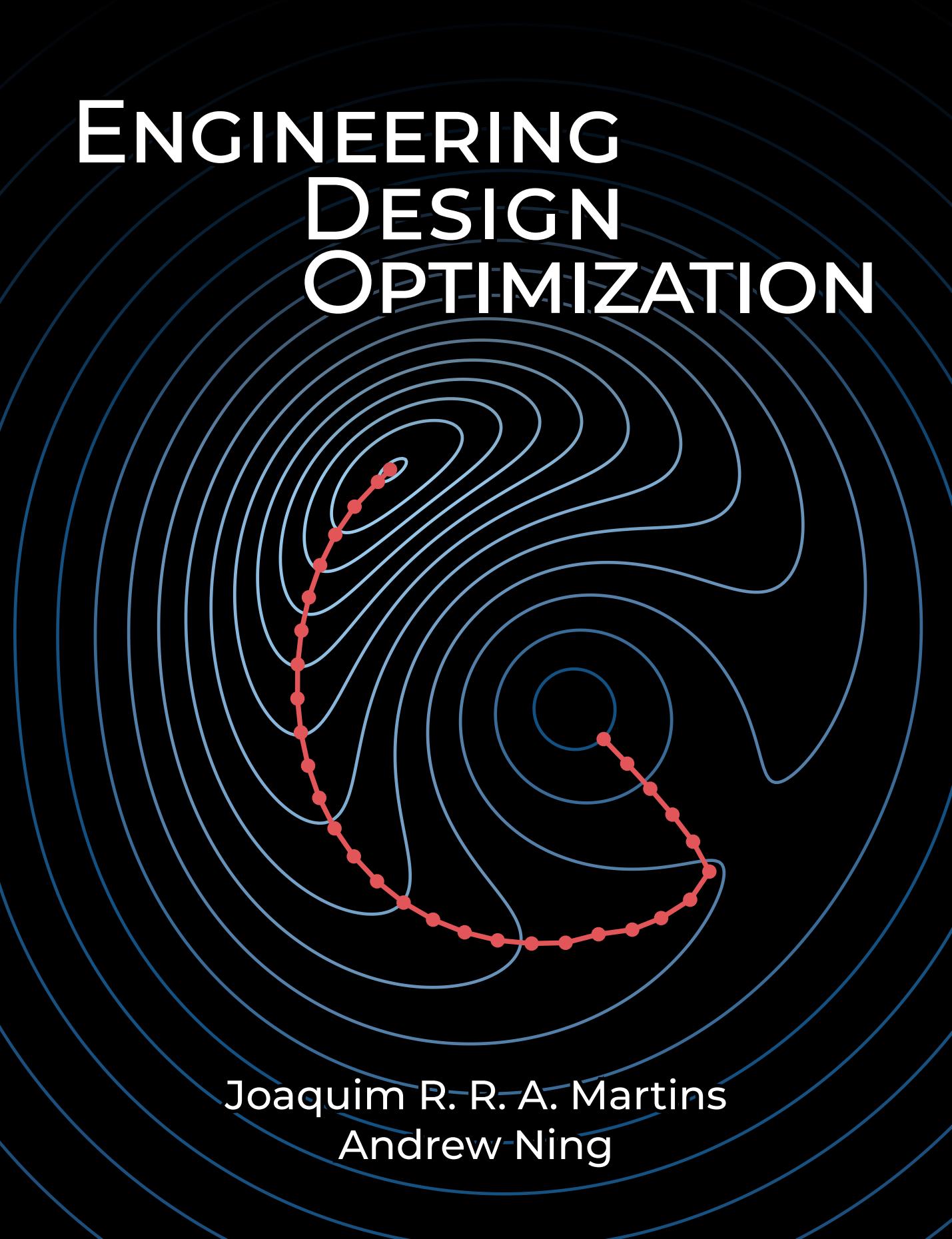


# ENGINEERING DESIGN OPTIMIZATION



Joaquim R. R. A. Martins  
Andrew Ning

# **Engineering Design Optimization**

JOAQUIM R. R. A. MARTINS

University of Michigan

ANDREW NING

Brigham Young University

This is a working **draft** that we are updating frequently. Once the book is finalized and published, we will continue to provide an electronic copy free of charge. If you have any suggestions or corrections, please email [jrram@umich.edu](mailto:jrram@umich.edu) or [aning@byu.edu](mailto:aning@byu.edu).

Draft compiled on Monday 25<sup>th</sup> January, 2021 at 11:47

**Copyright**

© 2020 Joaquim R. R. A. Martins and Andrew Ning. All rights reserved.

**Publication**

First electronic edition January 2020.

# Contents

---

Contents i

Preface v

Acknowledgements vii

## 1 Introduction 1

- 1.1 Design Optimization Process 2
- 1.2 Optimization Problem Formulation 5
- 1.3 Optimization Problem Classification 15
- 1.4 Optimization Algorithms 20
- 1.5 Selecting an Optimization Approach 25
- 1.6 Notation 27
- 1.7 Summary 27
- Problems 27

## 2 A Short History of Optimization 31

- 2.1 The First Problems: Optimizing Length and Area 31
- 2.2 Optimization Revolution: Derivatives and Calculus 32
- 2.3 The Birth of Optimization Algorithms 34
- 2.4 The Last Decades 37
- 2.5 Summary 41

## 3 Numerical Models and Solvers 42

- 3.1 Model Development for Analysis Versus Optimization 42
- 3.2 Modeling Process and Types of Errors 43
- 3.3 Numerical Models as Residual Equations 44
- 3.4 Discretization of Differential Equations 46
- 3.5 Numerical Errors 48
- 3.6 Rate of Convergence 54
- 3.7 Overview of Solvers 58
- 3.8 Newton-based Solvers 60
- 3.9 Models and the Optimization Problem 63
- 3.10 Summary 65

Problems 66

4 Unconstrained Gradient-Based Optimization 69

- 4.1 Fundamentals 70
- 4.2 Two Overall Approaches to Finding an Optimum 82
- 4.3 Line Search 84
- 4.4 Search Direction 95
- 4.5 Trust-region Methods 110
- 4.6 Summary 116
- Problems 117

5 Constrained Gradient-Based Optimization 122

- 5.1 Constrained Problem Formulation 123
- 5.2 Optimality Conditions 125
- 5.3 Penalty Methods 137
- 5.4 Sequential Quadratic Programming 148
- 5.5 Interior Point Methods 155
- 5.6 Merit Functions and Filters 160
- 5.7 Constraint Aggregation 163
- 5.8 Summary 164
- Problems 165

6 Computing Derivatives 172

- 6.1 Derivatives, Gradients, and Jacobians 172
- 6.2 Overview of Methods for Computing Derivatives 174
- 6.3 Symbolic Differentiation 175
- 6.4 Finite Differences 176
- 6.5 Complex Step 181
- 6.6 Algorithmic Differentiation 186
- 6.7 Implicit Analytic Methods—Direct and Adjoint 194
- 6.8 Sparse Jacobians and Graph Coloring 201
- 6.9 Unification of the Methods for Computing Derivatives 204
- 6.10 Summary 207
- Problems 208

7 Gradient-Free Optimization 211

- 7.1 Relevant Problem Characteristics 211
- 7.2 Classification of Gradient-Free Algorithms 214
- 7.3 Nelder–Mead Algorithm 217
- 7.4 DIRECT Algorithm 221
- 7.5 Genetic Algorithms 230
- 7.6 Particle Swarm Optimization 240
- 7.7 Summary 245

Problems	246
<b>8 Discrete Optimization</b>	249
8.1 Binary, Integer, and Discrete Variables	249
8.2 Techniques to Avoid Discrete Variables	251
8.3 Branch and Bound	252
8.4 Greedy Algorithms	259
8.5 Dynamic Programming	261
8.6 Simulated Annealing	269
8.7 Quantum Annealing	273
8.8 Binary Genetic Algorithms	273
8.9 Summary	273
Problems	274
<b>9 Multiobjective Optimization</b>	277
9.1 Multiple Objectives	277
9.2 Pareto Optimality	279
9.3 Solution Methods	280
9.4 Summary	291
Problems	291
<b>10 Surrogate-Based Optimization</b>	294
10.1 When to Use a Surrogate	294
10.2 Sampling	295
10.3 Constructing a Surrogate	298
10.4 Infill	304
10.5 Deep Neural Networks	306
10.6 Summary	312
Problems	313
<b>11 Convex Optimization</b>	316
11.1 Introduction	316
11.2 Linear Programming	318
11.3 Quadratic Programming:	320
11.4 Second-Order Cone Programming:	323
11.5 Disciplined Convex Optimization	324
11.6 Geometric Programming	325
11.7 Summary	328
Problems	328
<b>12 Optimization Under Uncertainty</b>	332
12.1 Introduction	332
12.2 Statistics Review	333

12.3	Robust Design	337
12.4	Reliability	341
12.5	Forward Propagation	343
12.6	Summary	354
	Problems	355
<b>13</b>	Multidisciplinary Design Optimization	359
13.1	Motivation	359
13.2	MDO Problem Representation	360
13.3	Multidisciplinary Models	360
13.4	Coupled Derivative Computation	372
13.5	Monolithic Architectures	375
13.6	Distributed Architectures	382
13.7	Summary	394
	Problems	395
<b>A</b>	Mathematics Review	397
A.1	Chain Rule, Partial Derivatives, and Total Derivatives	397
A.2	Vector and Matrix Norms	399
A.3	Matrix Multiplication	399
A.4	Matrix Types	402
A.5	Matrix Derivatives	403
A.6	Taylor Series Expansion	404
<b>B</b>	Linear Solvers	407
B.1	Direct Methods	407
B.2	Iterative Methods	409
<b>C</b>	Test Problems	413
C.1	Unconstrained Problems	413
C.2	Constrained Problems	419
	Bibliography	423
	Index	438

## Preface

---

Despite its usefulness, design optimization remains underused in industry. One of the reasons for this is the shortage of design optimization courses in undergraduate and graduate curricula. This is changing, as most top aerospace and mechanical engineering departments nowadays include at least one graduate-level course on numerical optimization. We have also seen design optimization increasingly used in industry, including the major aircraft manufacturers.

The usage of engineering in the title reflects the types of problems and algorithms we focus on, even though the methods are applicable beyond engineering. In contrast to explicit analytic mathematical functions, most engineering problems are implemented in complex multidisciplinary codes that involve implicit functions. Such problems might require hierarchical solvers and coupled derivative computation. Furthermore, many engineering problems involve many design variables with varying scales and many constraints, requiring scalable methods.

The target audience for this book is advanced undergraduate and beginning graduate students in science and engineering. No previous exposure to optimization is assumed. Knowledge of linear algebra, multivariable calculus, and numerical methods is helpful, although these subjects' core concepts are reviewed in the appendix. The content of the book spans approximately two semester-length university courses. Our approach is to start from the most general case problem and then explain some of the special cases. The first half of the book covers the fundamentals (along with an optional history chapter), whereas the second half, from Chapter 8 onwards, covers more specialized or advanced topics.

Our philosophy in the exposition is to provide a detailed enough explanation and analysis of optimization methods so that readers can implement a basic working version. The problems at the end of each chapter are designed to provide a gradual progression in difficulty and eventually require implementing the methods. Some of the problems are open-ended to encourage students to explore a given topic on their own. While we do not generally encourage readers to use their

implementations instead of existing tools for solving optimization problems, implementing a method is a useful exercise to understand the method and its behavior. A deeper understanding of these methods is useful for developers and researchers and for users who want to use numerical optimization more effectively. When discussing the various optimization techniques, we also explain how to avoid the potential pitfalls of using a particular method and how to use it more effectively. Practical tips are included throughout the book to alert the reader to common issues encountered in practical engineering design optimization and how to address them.

We have created a repository with code, data, templates, and examples as a supplementary resource for this book: <https://github.com/mdobook/resources>. Some of the end-of-chapter exercises refer to code or data from this repository.

## Acknowledgments

---

We are indebted to many students at our respective institutions that provided feedback on concepts, examples, and drafts of the manuscript. We wish to particularly thank Edmund Lee and Aaron Lu for translating many of our figures to a readable and precise format. We are also grateful to Judd Mehr for creating the initial draft for the mathematical review section of the appendix and to Max Opgenoord for sharing his thesis style file, on which the layout of this book is based.

Joaquim Martins and Andrew Ning

## Introduction

---

# 1

Optimization is a human instinct. People constantly seek to improve their lives and the systems that surround them. Optimization is intrinsic in biology, as exemplified by the evolution of species. Birds optimize their wings' shape in real time, and dogs have been shown to find optimal trajectories. Even more broadly, many laws of physics relate to optimization, such as the principle of minimum energy. As Leonhard Euler once wrote, "nothing at all takes place in the universe in which some rule of maximum or minimum does not appear."

Optimization is often used to mean improvement, but mathematically it is a much more precise concept: finding the *best* possible solution by changing variables that can be controlled, often subject to constraints. Optimization has a broad appeal because it is applicable in all domains and because we can all identify with a desire to make things better. Any problem where a decision needs to be made can be cast as an optimization problem.

While some simple optimization problems can be solved analytically, most practical problems of interest are too complex to be solved this way. The advent of numerical computing, together with the development of optimization algorithms, has enabled us to solve problems of increasing complexity.

Optimization problems occur in various areas, such as economics, political science, management, manufacturing, biology, physics, and engineering. A large segment of optimization applications focuses on *operations research*, which deals with problems such as deciding on the price of a product, setting up a distribution network, scheduling, or suggesting routes.

Another large segment of applications focuses on the design of engineering systems—the subject of this book. Design optimization problems abound in the various engineering disciplines, such as wing design in aerospace engineering, process control in chemical engineering, structural design in civil engineering, circuit design in electrical engineering, and mechanism design in mechanical engineering. Most engineering systems rarely work in isolation and are linked to other systems. This gave rise to the field of *multidisciplinary design optimization*.

(MDO), which applies numerical optimization techniques to the design of engineering systems that involve multiple disciplines.

In the remainder of this chapter, we start by explaining the design optimization process and contrasting it with the conventional design process (Section 1.1). Then we explain how to formulate optimization problems and the different types of problems that can arise (Section 1.2). Since design optimization problems involve functions of different types, these are also briefly discussed (Section 1.3). (A more detailed discussion of the numerical models used to compute these functions is deferred to Chapter 3.) We then provide an overview of the different optimization algorithms, highlighting the algorithms covered in this book and linking to the relevant section (Section 1.4). We connect algorithm types and problem types by providing guidelines for selecting the right algorithm for a given problem (Section 1.5). Finally, we introduce the notation used throughout the book (Section 1.6).

By the end of this chapter you should be able to:

1. Understand the design optimization process.
2. Formulate an optimization problem.
3. Identify key characteristics to classify optimization problems and optimization algorithms.
4. Recognize some salient characteristics in selecting an appropriate algorithm.

## 1.1 Design Optimization Process

Engineering design is an iterative process that engineers follow to develop a product that accomplishes a given task. For any product beyond a certain complexity, this process involves teams of engineers and multiple stages with many iterative loops that could be nested. The engineering teams are formed to tackle different aspects of the product at different stages.

The design process can be divided into the sequence of phases shown in Fig. 1.1. Before the design process begins, we must determine the requirements and specifications. This might involve market research, an analysis of current similar designs, and interviews with potential customers. In the conceptual design phase, various concepts for the system are generated and considered. Because this phase should be

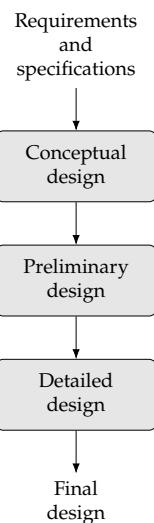


Figure 1.1: Design phases.

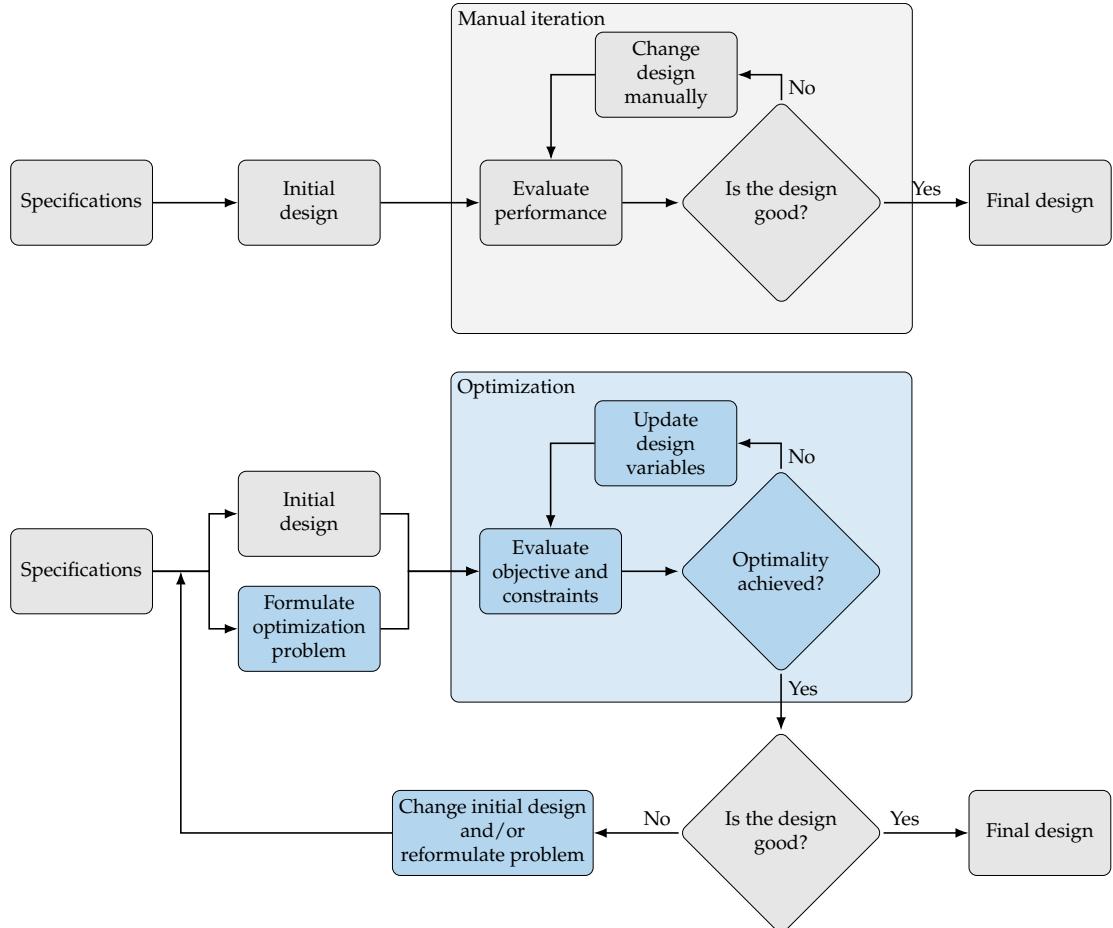
short, it usually relies on simplified models and human intuition. For more complicated systems, the various subsystems are identified. In the preliminary design phase, a chosen concept and subsystems are refined by using better models to guide changes in the design, and the performance expectations are set. The detailed design phase seeks to complete the design down to every detail so that it can finally be manufactured. All of these phases require iteration within themselves. When severe issues are identified, it may be necessary to “go back to the drawing board” and regress to an earlier phase. This is just a high-level view; in practical design, each phase may require multiple iterative processes.

Design optimization is a tool that can be used to replace an iterative design process to accelerate the design cycle and obtain better results. To understand the role of design optimization, consider a simplified version of the conventional engineering design process with only one iterative loop, as shown in Fig. 1.2 (top). In this process, engineers make decisions at every stage based on intuition and background knowledge.

Each of the conventional design process steps includes human decisions that are either challenging or impossible to program into computer code. Determining specifications for the product require engineers to define the problem and do background research. The design cycle must start with an initial design, which can be based on past designs or a new idea. In the conventional design process, this initial design is analyzed in some way to evaluate its performance. This could involve numerical modeling or actual building and testing. Engineers then evaluate the design and decide whether it is good enough or not based on the results.\* If the answer is no—which is likely to be the case for at least the first few iterations—the engineer will change the design based on intuition, experience, or trade studies. When the design is satisfactory, the engineer will arrive at the final design.

The design optimization process can be represented using a similar flow diagram, as shown in Fig. 1.2 (bottom). The determination of the specification and the initial design are no different from the conventional design process. However, design optimization requires a formal formulation of the optimization problem that includes the design variables that are to be changed, the objective to be minimized, and the constraints that need to be satisfied. The evaluation of the design is strictly based on numerical values for the objective and constraints. When a rigorous optimization algorithm is used, the decision to finalize the design is only made when the current design satisfies the optimality conditions that ensure that no other design “close by” is better. The design changes are made automatically by the optimization algorithm

\*The evaluation of a given design is often just called the *analysis*.



and do not require intervention from the designer.

This automated process does not usually provide a “push-button” solution; it requires human intervention and expertise (often more expertise than in the traditional process). Human decisions are still needed in the design optimization process. Before running an optimization, in addition to determining the specifications and initial design, engineers need to formulate the design problem. This requires expertise in both the subject area and in numerical optimization. The designer must decide what the objective is, which parameters can be changed, and which constraints must be enforced.

After running the optimization, engineers must assess the design because it is unlikely that a valid and practical design is obtained after the first time a formulation is developed. After evaluating the optimal design, engineers might decide to reformulate the optimization problem by changing the objective function, adding or removing constraints,

**Figure 1.2:** Conventional (top) versus design optimization process (bottom).

or changing the set of design variables. Engineers might also decide to increase the models' fidelity if they fail to consider critical physical phenomena or decrease the fidelity if the models are too expensive to evaluate in an optimization iteration.

In addition to assessing the optimization results, post-optimality studies are often performed to interpret the optimal design and the design trends. This might be done by performing parameter studies, where design variables or other parameters are varied to quantify their effect on the objective and constraints. Validation of the result can be done by evaluating the design with higher-fidelity simulation tools, by performing experiments, or both. It is also possible to compute post-optimality sensitivities to evaluate which design variables are the most influential or which constraints drive the design. These sensitivities can inform where engineers might best allocate resources to alleviate the driving constraints in future designs.

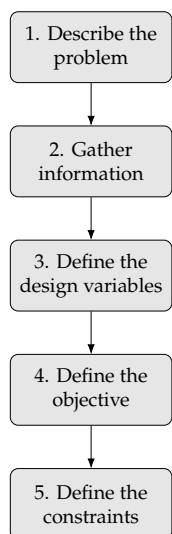
Design optimization can be used in any of the design phases shown in Fig. 1.1, where each phase could involve running one or more design optimizations. In addition to increasing the system performance and reducing the design time, design optimization also decreases the uncertainty at any given time compared to the conventional design process (Fig. 1.3). Considering multiple disciplines or components using MDO amplifies these same favorable trends.

## 1.2 Optimization Problem Formulation

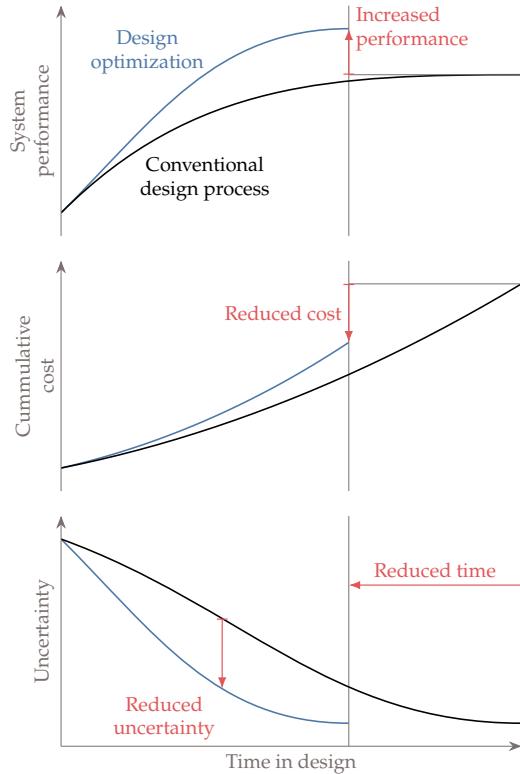
The design optimization process requires the designer to translate their intent to a mathematical statement that can then be solved by an optimization algorithm. Developing this statement has the added benefit that it helps the designer better understand the problem. Being methodical in the optimization problem formulation is vital because *the optimizer tends to exploit any weaknesses you might have in your formulation or model*. An inadequate problem formulation can either cause the optimization to fail or to converge to a mathematical optimum that is undesirable or unrealistic from an engineering point of view—the proverbial “right answer to the wrong question”.

To formulate design optimization problems, we follow the procedure outlined in Fig. 1.4. The first step requires writing a description of the design problem, including a description of the system, and a statement of all the goals and requirements. At this point, the description does not necessarily involve optimization concepts and is often vague.

The next step is to gather and much data and information as possible about the problem. Some information is already specified in



**Figure 1.4:** Optimization problem formulation steps.



**Figure 1.3:** Compared to the conventional design process, MDO increases the system performance, decreases the design time, reduces the total cost, and reduces the uncertainty at a given point in time.

the problem statement, but more research is usually required to find all the relevant data on the performance requirements and expectations. Raw data might need to be processed and organized to gather the information required for the design problem. The more familiar practitioners are with the problem, the better prepared they will be to develop a sound formulation to identify eventual issues in the solutions.

At this stage, it is also essential to identify the analysis procedure and gather information on that as well. The analysis might consist of a simple model or a set of elaborate tools. All the possible inputs and outputs of the analysis should be identified, and its limitations should be understood. The computational time for the analysis needs to be considered because optimization requires repeated analysis.

It is usually impossible to learn everything about the problem before proceeding to the next steps where we define the design variables, objective, and constraints. Therefore, information gathering and refinement is an ongoing process in the problem formulation.

### 1.2.1 Design Variables

The next step is to identify the variables that describe the system, the *design variables*, which we represent by the vector:

$$\mathbf{x} = [x_1, x_2, \dots, x_{n_x}]^T. \quad (1.1)$$

This vector defines a given design, so different vectors  $\mathbf{x}$  correspond to different designs. The number of variables,  $n_x$ , determines the problem's dimensionality and is also referred to as the design degrees of freedom.

The design variables must not depend on each other, or any other parameter, and the optimizer must be free to choose the components of  $\mathbf{x}$  independently. This means that in the analysis of a given design, they must be input parameters that remain fixed throughout the analysis process. Otherwise, the optimizer will not have absolute control of the design variables, and the underlying mathematical assumptions break down. Another possible pitfall is to define a design variable that happens to be a linear combination of other variables, which results in an ill-defined optimization problem with an infinite number of combinations of design variable values that correspond to the same design.

The choice of variables is usually not unique. For example, a square shape can be parametrized by the length of its side or by its area, and different unit systems can be used. The choice of units affects the problem's scaling, but not the functional form of the problem.

The choice of design variables can affect the functional form of the objective and constraints. For example, a nonlinear problem could be converted to a linear problem through a change of variables. It is also possible to introduce or eliminate discontinuities through the choice of design variables.

A given set of design variable values defines the system's design, but whether this system satisfies all the requirements is a separate question that will be addressed with the constraints in a later step. However, it is possible and advisable to define the space of allowable values for the design variables based on the design problem specifications and physical limitations.

The first consideration in the definition of the allowable design variable values is whether the design variables are *continuous* or *discrete*. The continuous design variables are real numbers that are allowed to vary continuously within a specified range with no gaps, which we write as

$$\underline{x}_i \leq x_i \leq \bar{x}_i; \quad i = 1, \dots, n_x, \quad (1.2)$$

where  $\underline{x}$  and  $\bar{x}$  are lower and upper bounds on the design variables, respectively. These are also known as side constraints. Some design variables may be unbounded or only bounded on one side.

We distinguish the design variable bounds from constraints because the optimizer has direct control over their values, and they benefit from a different numerical treatment when solving an optimization problem. When defining these bounds, we must take care not to unnecessarily constrain the design space, which would prevent the optimizer from achieving a better design that is realizable. A smaller allowable range in the design variable values should make the optimization easier. However, design variable bounds should be based on actual physical constraints instead of being artificially limited. An example of a physical constraint is a lower bound on structural thickness in a weight minimization problem, where otherwise, the optimizer will discover that negative sizes yield negative weight. Whenever a design variable converges to the bound at the optimum, you should reconsider the reasoning for that bound and make sure it is valid. This is because designers sometimes set bounds that unnecessarily limit the optimization from obtaining a better objective.

In a continuous optimization problem, all design variables must be continuous.<sup>†</sup> Most of this book focuses on algorithms that assume continuous design variables.

When one or more variables are discrete, we have a discrete optimization problem. The discrete case occurs when one or more variables are allowed to have discrete values, which can be either real or integer. An example of discrete design variables is structural sizing, where only components of specific thicknesses or cross-sectional areas are available. Integer design variables are a special case of discrete variables where the values are integers, like the number of wheels in a vehicle. Optimization algorithms that handle discrete variables are discussed in Chapter 8.

At the formulation stage, we should strive to list as many independent design variables as possible. However, it is advisable to start with a small set of variables when solving the problem for the first time and then gradually expand the set of design variables.

If the optimization algorithm requires it, an initial design should be specified by assigning values to each design variable. While it is easy to assign values within the bounds, it might not be as easy to ensure that the initial design satisfies the constraints. This is not an issue for most optimization algorithms, but some require starting with a feasible

<sup>†</sup>This is not to be confused with the continuity of the objective and functions, which we discuss in Section 1.3.

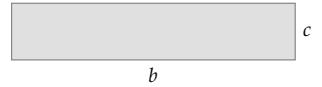
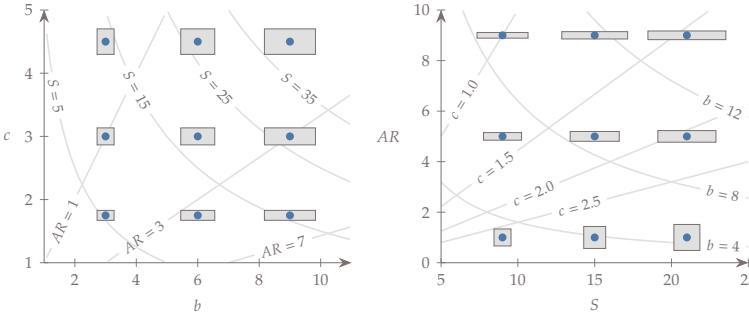
design.

---

**Example 1.1:** Design variables for wing design.

---

Consider a wing design problem where the wing planform shape is rectangular. The planform could be parametrized by the span ( $b$ ) and the chord ( $c$ ) as seen in Fig. 1.5, so that  $x = [b, c]^T$ . However, this choice is not unique.



**Figure 1.5:** Wing span ( $b$ ) and chord ( $c$ ).

Two other variables are often used in aircraft design: wing area ( $S$ ) and wing aspect ratio ( $AR$ ), also shown in the figure. Because these variables are not independent ( $S = bc$  and  $AR = b^2/S$ ), we cannot just add them to the set of design variables. Instead, we must pick any two variables out of the four to parametrize the design because we have four possible variables and two dependency relationships.

For this wing, the variables must be positive to be physically meaningful, so we must remember to explicitly bound these variables to be greater than zero in an optimization. The variables should be bound from below by small positive values because numerical models are probably not prepared to take zero values. No upper bound is needed unless the optimization algorithm requires it.

---

### 1.2.2 Objective Function

To find the best design, we need a quantifiable criterion to determine if one design is better than another—the *objective function*. The objective function must be a scalar that is computable for a given design variable vector  $x$ . The objective function can be minimized or maximized, depending on the problem. For example, a designer might want to minimize the weight or cost of a given structure. An example of a function to be maximized could be the range of a vehicle.

The convention adopted in this book is that the objective function,  $f$ , is to be *minimized*. This convention does not prevent us from maximizing

**Figure 1.6:** Wing design space for two different sets of design variables,  $x = [b, c]^T$  and  $x = [S, AR]^T$ .

a function, since we can reformulate it as a minimization problem by finding the minimum of the negative of  $f$  and then changing the sign, that is:

$$\max[f(x)] = -\min[-f(x)]. \quad (1.3)$$

This transformation is illustrated in Fig. 1.7.<sup>‡</sup>

The computation of the objective function is done through a numerical model whose complexity can range from a simple explicit equation to a system of coupled implicit models (more on this in Chapter 3).

The choice of objective function is crucial for successful design optimization. If the function does not represent the true intent of the designer, it does not matter how precisely the function and its optimum point is computed—the mathematical optimum will be non-optimal from the engineering point of view. A bad choice of objective function is a common mistake in design optimization.

The choice of objective function is not always obvious. For example, minimizing the weight of a vehicle might sound like a good idea, but this might result in a vehicle that is too expensive to manufacture. In this case, manufacturing cost would probably be a better objective. However, there is a tradeoff between manufacturing cost and the efficiency of the vehicle. It might not be obvious which of these objectives is the most appropriate one because this trade depends on customer preferences. This issue motivates *multiobjective optimization*, which is the subject of Chapter 9. Multiobjective optimization does not yield a single design but rather a range of designs that settle for different tradeoffs between the objectives.

Experimenting with different objectives should be part of the design exploration process (this is represented by the outer loop in the design optimization process in Fig. 1.2). Results from optimizing the “wrong” objective can still yield insights into the design tradeoffs and trends for the system at hand.

---

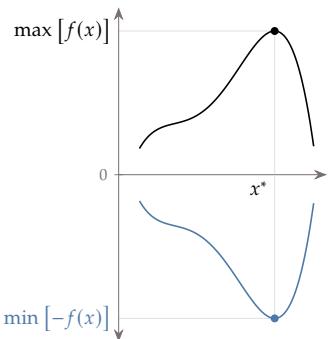
**Example 1.2:** Objective function for wing design.
 

---

Let us consider the appropriate objective function for Ex. 1.1. A common objective for a wing is to minimize drag. However, this does not take into account the propulsive efficiency, which is strongly affected by speed. A better objective might be to minimize the required power, which balances drag and propulsive efficiency.

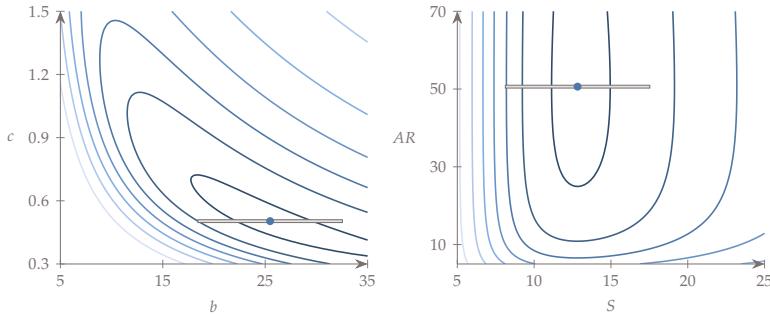
The contours for the required power are shown in Fig. 1.8 for the two choices of design variable sets discussed in Ex. 1.1. We can locate the minimum graphically. While the two optimum solutions are the same, the shapes of the objective function contours are different. In this case, using aspect ratio

<sup>‡</sup>Inverting the function  $(1/f)$  is also a possible way to turn a maximization problem into a minimization one, but is generally less desirable as it alters the scale of the problem and could introduce a divide-by-zero problem.



**Figure 1.7:** A maximization problem can be transformed into an equivalent minimization one.

and wing area simplifies the relationship between the design variables and the objective by aligning the two main curvature trends with each design variable.



**Figure 1.8:** Required power contours for two different choices of design variable sets. The optimal wing is the same for both cases, but the functional form of the objective is simplified in the bottom one.

In this case, the optimal wing has an aspect ratio that is much higher than typically seen in aircraft or birds. While the high aspect ratio increases aerodynamic efficiency, it adversely affects the structural strength, which we did not consider here. Thus, as in most engineering problems, we need to add constraints.

---

A better metric might be to minimize the required power, which naturally balances drag and propulsive efficiency. The wing area affects how fast it must fly to generate sufficient lift, which in turn affects the efficiency of the propulsion system. Contours for this objective, the optimal span and chord (denoted by the dot), and the shape of the optimal wing are shown in Fig. 1.8.

While we can sometimes visualize the variation of the objective function as in Ex. 1.2, this is not possible for problems with more design variables or more computationally demanding function evaluations. This motivates numerical optimization algorithms, which aim to find the minimum in a multidimensional design space using as few function evaluations as possible.

### 1.2.3 Constraints

The vast majority of practical design optimization problems require the enforcement of constraints. These are functions of the design variables that we want to restrict in some way. Like the objective function, constraints are computed through a model whose complexity can vary widely.

When we restrict a function to being equal to a fixed value, we call this an *equality constraint*, denoted by  $h(x) = 0$ . When the function is required to be less than or equal to a certain value, we have an

*inequality constraint*, denoted by  $g(x) \leq 0$ .<sup>§</sup> While we use “less or equal” by convention, you should be aware that some other texts and software programs use “greater or equal” instead. There is no loss of generality with either convention, as we can always multiply the constraint by  $-1$  to convert between the two.

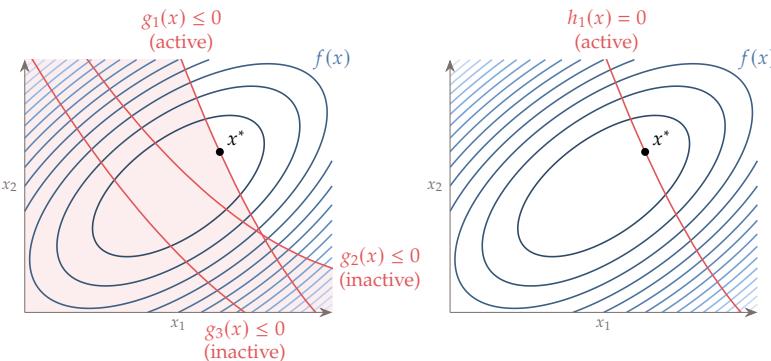
**Tip 1.3:** Check the inequality convention.

When using optimization software, do not forget to check the convention for the inequality constraints (is it “less than” or “greater than”) and convert your constraints as needed.

<sup>§</sup>A strict inequality,  $g(x) < 0$ , is never used because then  $x$  could be arbitrarily close to the equality. Since the optimum is at  $g = 0$  for an active constraint, the exact solution would then be ill-defined from a mathematical perspective. Also, the difference is not meaningful when using finite-precision arithmetic (which is always the case when using a computer).

Some texts omit the equality constraints without loss of generality because an equality constraint can be replaced by two inequality constraints. More specifically, an equality constraint,  $h(x) = 0$ , is equivalent to two inequality constraints,  $g(x) \geq 0$  and  $g(x) \leq 0$ .

Inequality constraints can be *active* or *inactive* at the optimum point. Active means that  $g(x^*) = 0$ , whereas inactive means  $g(x^*) < 0$ . If inactive, then the corresponding constraint could have been removed from the problem with no change in its solution, as illustrated in Fig. 1.9. In the general case, however, it is difficult to know in advance which constraints are active or not at the optimum. Constrained optimization is the subject of Chapter 5.



**Figure 1.9:** Example of two-dimensional problem with one active and two inactive inequality constraints (left). The red highlighted area indicates regions that are *infeasible* (i.e., the constraints are violated). If we only had the active single equality constraint in the formulation, we would obtain the same result (right).

It is possible to over-constrain the problem such that there is no solution. This can happen due to a programming error but can also occur at the problem formulation stage. For more complicated design problems, it might not be possible to satisfy all the specified constraints, even if they seem to make sense. When this happens, constraints have to be relaxed or removed.

The problem must not be over-constrained, or else there is no feasible region in the design space over which the function can be minimized. Thus, the number of independent equality constraints must be less or equal to the number of design variables ( $n_h \leq n_x$ ). There is no limit on the number of inequality constraints. However, they must be such that there is a feasible region, and the number of active constraints plus the equality constraints must still be less or equal than the number of design variables.

One common issue in optimization problem formulation is distinguishing objectives from constraints. For example, we might be tempted to minimize the stress in a structure, but this would inevitably result in an overdesigned heavy structure. Instead, we might want minimum weight (or cost) with sufficient safety factors on stress, which can be enforced by an inequality constraint.

Most engineering problems require constraints—often a large number of them. While constraints may at first appear limiting, they are what enable the optimizer to find useful solutions.

---

**Example 1.4:** Constraints for wing design.
 

---

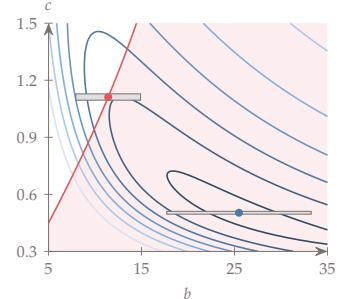
We now add a design constraint for the power minimization problem of Ex. 1.2. The unconstrained optimal wing has unrealistically high aspect ratios because we did not include structural considerations. If we add an inequality constraint on the bending stress at the root of the wing for a fixed amount of material, we get the curve and feasible region shown in Fig. 1.10. The unconstrained optimum violates this constraint. The constrained optimum results in a lower span and higher chord, and the constraint is active.

---

As previously mentioned, it is not possible in general to visualize the design space as shown in Ex. 1.2 and obtain the solution graphically. In addition to the possibility of a large number of design variables and computationally expensive objective function evaluations, we now add the possibility of a large number of constraints that are also expensive to evaluate. Again, this is further motivation for the optimization techniques covered in this book.

#### 1.2.4 Optimization Problem Statement

Now that we have discussed the definition of design variables, objective function, and constraints, we can put them all together in an optimization problem statement. In words, this statement is: *Minimize the objective function by varying the design variables subject to the constraints and design variable bounds.* Mathematically, we write this as:<sup>¶</sup>



**Figure 1.10:** Minimum power wing with a constraint on bending stress compared to the unconstrained case.

<sup>¶</sup>Instead of “by varying”, some textbooks write “with respect to”.

$$\begin{aligned}
 & \text{minimize} && f(x) \\
 & \text{by varying} && \underline{x}_i \leq x_i \leq \bar{x}_i \quad i = 1, \dots, n_x \\
 & \text{subject to} && g_j(x) \leq 0 \quad j = 1, \dots, n_g \\
 & && h_k(x) = 0 \quad k = 1, \dots, n_h
 \end{aligned} \tag{1.4}$$

While this is the standard formulation used in this book, other books and software manuals might differ from this. For example, they might use different symbols, use “greater or equal than” for the inequality constraint, or maximize instead of minimizing. In any case, it is possible to convert between standard formulations to get equivalent problems.

All single objective, continuous optimization problems can be written in this form. Although our target applications are engineering design problems, many other problems can be stated in this form, and thus, the methods covered in this book can be used to solve those problems.

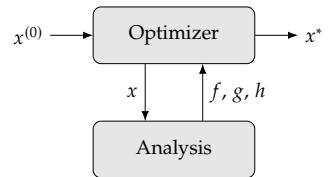
The values of the objective and constraint functions for a given set of design variables are computed through the analysis, which consists of one or more numerical models. The analysis must be fully automatic so that multiple optimization cycles can be completed without human intervention, as shown in Fig. 1.11. The optimizer usually requires an initial design  $x^{(0)}$  and then queries the analysis for a sequence of designs until it finds the optimum design,  $x^*$ .

**Tip 1.5:** Using an optimization software package.

The setup of an optimization problem varies depending on the particular software package, so read the documentation carefully. Most optimization software requires you to define the objective and constraints as *callback functions*. These are passed to the optimizer, which will call them back as needed during the optimization process. The functions take the design variable values as inputs and output the function values, as shown in Fig. 1.11. Study the software documentation for the details of how to use it.<sup>¶</sup> To make sure you understand how to use a given optimization package, test it on simple problems for which you know the solution first (see Prob. 1.5).

When the optimizer queries the analysis for a given  $x$ , the constraints do not have to be feasible. The optimizer is responsible for changing  $x$  so that the constraints are satisfied.

The objective and constraint functions must depend on the design variables; if a function does not depend on any variable in the whole domain, it can be ignored and should not appear in the problem statement.



**Figure 1.11:** The analysis computes the objective ( $f$ ) and constraint values ( $g, h$ ) for a given set of design variables ( $x$ ).

<sup>¶</sup> Possible software includes: `fmincon` in Matlab, `scipy.optimize.minimize` with the `SLSQP` method in Python, `Optim.jl` with the `IPNewton` method in Julia, and the Solver add-in in Microsoft Excel.

Ideally,  $f$ ,  $g$ , and  $h$  should be computable for all values of  $x$  that make physical sense. Lower and upper design variable bounds should be set to avoid non-physical designs as much as possible. Even after taking this precaution, models in the analysis sometimes fail to provide a solution. A good optimizer can handle such eventualities gracefully.

There are some mathematical transformations that do not change the solution of the optimization problem (Eq. 1.4). Multiplying either the objective and constraints by a constant does not change the optimal design; it only changes the optimum objective value. Adding a constant to the objective does not change the solution, but adding a constant to any constraint changes the feasible space and can change the optimal design.

Determining an appropriate set of design variables, objective, and constraints is a crucial aspect of the outer loop shown in Fig. 1.2, which requires human expertise in engineering design and numerical optimization.

---

**Tip 1.6:** Ease into the problem.

It is tempting to set up the full problem and attempt to solve it right away. This rarely works, especially for a new problem. Before attempting any optimization, you should run the analysis models and explore the solution space manually. Particularly if using gradient-based methods, it helps plot the output functions across multiple input sweeps to assess if the numerical outputs display the expected behavior and smoothness. Instead of solving the full problem, ease into it by setting up the simplest subproblem possible. If the function evaluations are costly, consider using computational models that are less costly (but still representative). It is advisable to start by solving a subproblem with a small set of variables and then gradually expanding it. Removing some constraints has to be done more carefully because it might result in an ill-defined problem. For multidisciplinary problems, you should run optimizations with each component separately before attempting to solve the coupled problem.

Solving simple problems for which you know the answer (or at least problems for which you know the trends) helps identify any issues with the models and problem formulation. Solving a sequence of increasingly complicated problems gradually builds an understanding of how to solve the optimization problem and interpret its results.

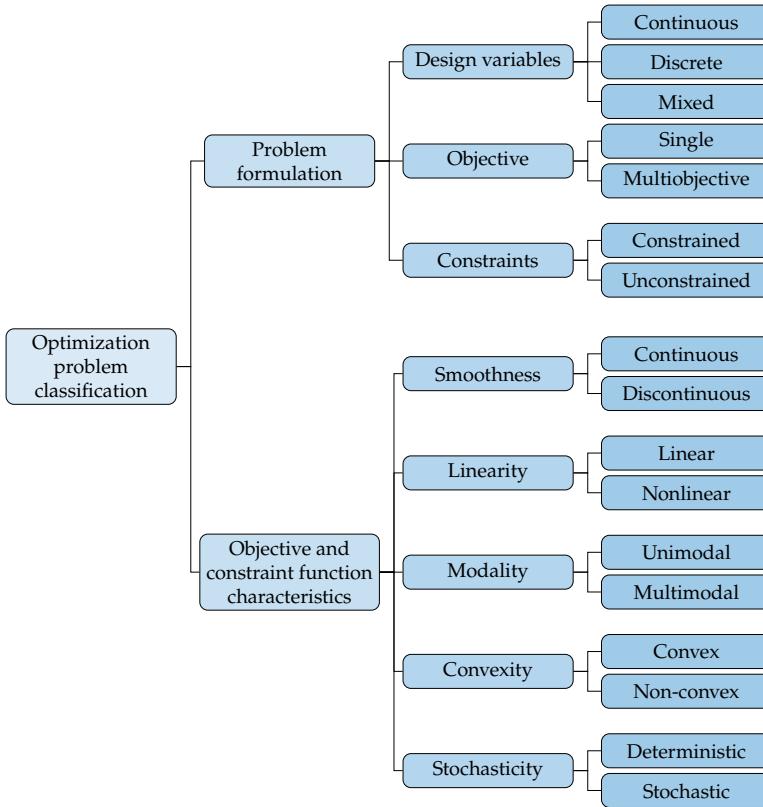
---

### 1.3 Optimization Problem Classification

To choose the most appropriate optimization algorithm for solving a given optimization problem, we must classify the optimization prob-

lem and know how its attributes affect the efficacy and suitability of the available optimization algorithms. This is important because no optimization algorithm is efficient or even appropriate for all types of problems.

We classify optimization problems based on two main aspects: the problem formulation and the characteristics of the objective and constraint functions, as shown in Fig. 1.12.



**Figure 1.12:** Optimization problems can be classified by attributes associated with the different aspects of the problem. The two main aspects are the problem formulation and the objective and constraint function characteristics.

The classification based on problem formulation was already discussed in Section 1.2. The design variables can be either discrete or continuous. Most of this book assumes continuous design variables, but Chapter 8 provides an introduction to discrete optimization. When the design variables include both discrete and continuous variables, the problem is said to be *mixed*. Most of the book assumes a single objective function, but we explain how to solve multiobjective problems in Chapter 9. Finally, as previously mentioned, unconstrained problems are rare in engineering design optimization. However, we explain unconstrained optimization algorithms (Chapter 4) because

they provide the foundations for constrained optimization algorithms (Chapter 5).

The characteristics of the objective and constraint functions also determine the type of optimization problem at hand and ultimately limit the type of optimization algorithm that is appropriate to solve the optimization problem.

In this section, we will view the function as a “black box”, that is, a computation for which we only see inputs (including the design variables) and outputs (including objective and constraints), as illustrated in Fig. 1.13.

When dealing with black-box models, there is limited or no understanding of the modeling and numerical solution process used to obtain the function values. We discuss the types of models and how to solve them in Chapter 3, but here we can still characterize the functions based purely on their outputs.

The black-box view is common in real-world applications. This might be because the source code is not provided, the modeling methods are not described, or simply because the user does not bother to understand them.

In the remainder of this section, we discuss the attributes of objectives and constraints shown in Fig. 1.12. Strictly speaking, many of these attributes cannot typically be identified from a black-box model. For example, while the model may appear smooth, we cannot know that it is smooth everywhere without a more detailed inspection. However, for this discussion, we assume that the black box’s outputs can be exhaustively explored so that these characteristics can be identified.

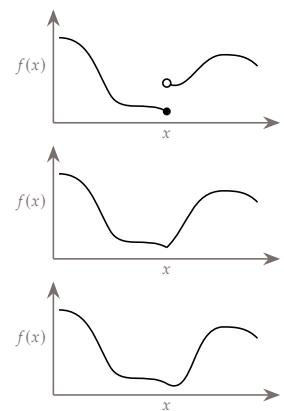
### 1.3.1 Smoothness

The degree of function smoothness with respect to variations in the design variables depends on the continuity of the function values and their derivatives. When the value of the function varies continuously, the function is said to be  $C^0$  continuous. If the first derivatives also vary continuously, then the function is  $C^1$  continuous, and so on. A function is smooth when the derivatives of all orders vary continuously everywhere in its domain. Function smoothness with respect to continuous design variables affects what type of optimization algorithm can be used. Figure 1.14 shows one-dimensional examples for a discontinuous,  $C^0$  function, and  $C^1$  function.

As we will see later, discontinuities in the function value or derivatives limit the type of optimization algorithm that can be used because some algorithms assume  $C^0$ ,  $C^1$ , and even  $C^2$  continuity. In practice,



**Figure 1.13:** A model is considered a black box when we only see its inputs and outputs.



**Figure 1.14:** Discontinuous function (top),  $C^0$  continuous function (middle), and  $C^1$  continuous function (bottom).

these algorithms usually still work with functions that have only a few discontinuities that are located away from the optimum.

### 1.3.2 Linearity

The functions of interest could be linear or nonlinear. When both the objective and constraint functions are linear, the optimization problem is known as a linear optimization problem. These problems are easier to solve than general nonlinear ones, and there are entire books and courses dedicated to the subject. The first numerical optimization algorithms were developed to solve linear optimization problems, and there are many applications in operations research (see Chapter 2). An example of a linear optimization problem is shown in Fig. 1.15.

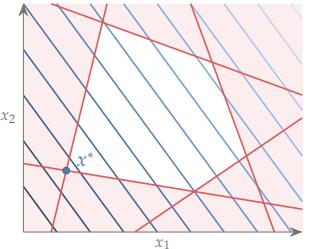
When the objective function is quadratic, and the constraints are linear, we have a quadratic optimization problem, which is another type of problem for which specialized solution methods exist.<sup>\*\*</sup> Linear optimization and quadratic optimization are covered in Sections 11.2 and 11.3.

While many problems can be formulated as linear or quadratic problems, most engineering design problems are nonlinear. However, it is common to have at least a subset of constraints that are linear, and some general nonlinear optimization algorithms take advantage of the techniques developed to solve linear and quadratic problems.

### 1.3.3 Multimodality and Convexity

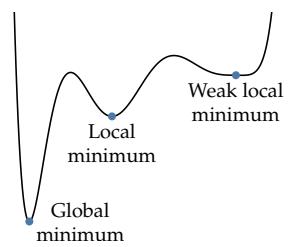
Functions can be either unimodal or multimodal. Unimodal functions have a single minimum, while multimodal functions have multiple minima. When we find a minimum without knowledge of whether the function is unimodal or not, we can only say that it is a *local minimum*, that is, this point is better than any point within a small neighborhood. When we know that a local minimum is the best in the whole domain (because we somehow know that the function is unimodal), then this is also the *global minimum*, as illustrated in Fig. 1.16. Sometimes, the function might be flat around the minimum, in which case we have a *weak minimum*.

For functions involving more complicated numerical models, it is usually impossible to prove that the function is unimodal. Proving that such a function is unimodal would require evaluating the function at every point in the domain, which is computationally prohibitive. However, it is much easier to prove multimodality—we just need to find two distinct local minima.



**Figure 1.15:** Example of a linear optimization problem in two dimensions.

<sup>\*\*</sup>Historically, optimization problems were referred to as “programming” problems, so much of the existing literature refers to these as “linear programming” and “quadratic programming”.



**Figure 1.16:** Types of minima.

Just because a function is complicated or the design space has many dimensions, it does not mean that the function is multimodal. By default, we should not assume that a given function is either unimodal or multimodal. As we explore the problem and solve it starting from different points or using different optimizers, there are two main possibilities. One possibility is that we find more than one minimum, thus proving that the function is multimodal. The other possibility is that the optimization consistently converges to the same optimum. In this case, we can become increasingly confident that the function is unimodal with every new optimization that converges to the same optimum.

Often, we need not be too concerned about the possibility of multiple local minima. From an engineering design point of view, achieving a local optimum that is better than the initial design is already a useful result.

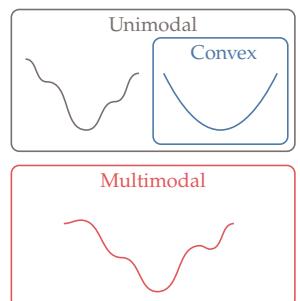
Convexity is a concept related to multimodality. A function is convex if all line segments connecting any two points in the function lies above the function and never intersect it. Convex functions are always unimodal. Also, all multimodal functions are non-convex, but not all unimodal functions are convex (see Fig. 1.17).

Convex optimization seeks to minimize convex functions over convex sets. Like linear optimization, convex optimization is another subfield of numerical optimization with many applications. When the objective and constraints are convex functions, we can use specialized formulations and algorithms that are much more efficient than general nonlinear algorithms to find the global optimum, as explained in Chapter 11.

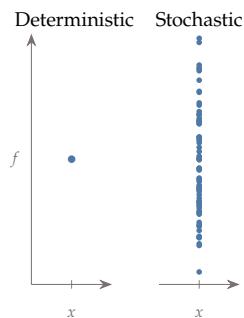
### 1.3.4 Deterministic Versus Stochastic

Some functions are inherently stochastic. A stochastic model will yield different function values for repeated evaluations with the same input (Fig. 1.18). For example, the numerical value from a roll of dice is a stochastic function.

Stochasticity can also arise from deterministic models when the inputs are subject to *uncertainty*. The input variables are then described as probability distributions, and their uncertainties need to be propagated through the model. For example, the bending stress in a beam may follow a deterministic model, but the beam's geometric properties may have uncertainty due to manufacturing deviations. For most of this text, we assume that functions are deterministic except in Chapter 12, where we provide an overview of this latter form of stochasticity under



**Figure 1.17:** Multimodal functions have multiple minima, while unimodal functions have only one minimum. All multimodal functions are non-convex, but not all unimodal functions are convex.



**Figure 1.18:** Deterministic function yield the same output when evaluated repeatedly for the same input, while stochastic functions do not.

the topic of optimization under uncertainty.

## 1.4 Optimization Algorithms

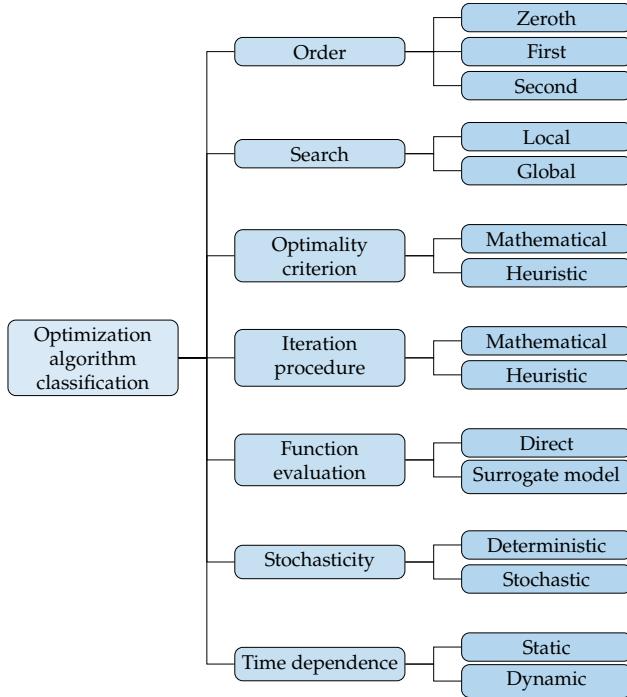
No single optimization algorithm is effective or even appropriate for all possible optimization problems. This is why it is important to understand the problem before deciding which optimization algorithm to use. By “effective” algorithm, we mean that the algorithm is capable of solving the problem at all, and secondly, it does so reliably and efficiently. Fig. 1.19 lists the attributes for the classification of optimization algorithms, which we discuss in more detail below. These attributes are often amalgamated, but they are independent and any combination is possible. In this text, we cover a wide variety of optimization algorithms corresponding to several of these combinations. However, this overview still does not cover a wide variety of specialized algorithms designed to solve specific problems where a particular structure can be exploited.

When multiple disciplinary models are involved, we also need to consider how the models are coupled, solved, and integrated with the optimizer. These considerations lead to different MDO *architectures*, which may involve multiple levels of optimization problems. Coupled models are introduced in Section 13.3, while MDO architectures are covered in Chapter 13.

### 1.4.1 Order of Information

At the minimum, an optimization algorithm requires users to provide the models that compute the objective and constraint values for any given set of allowed design variables. We call algorithms that use just these function values *gradient-free* algorithms (also known as derivative-free or zeroth-order algorithms). We cover a selection of these algorithms in Chapter 7. The advantage of gradient-free algorithms is that the optimization is easier to setup because they do not need additional computations other than what the models for the objective and constraints already provide.

*Gradient-based* algorithms use gradients of both the objective and constraint functions with respect to the design variables. We first cover gradient-based algorithms for unconstrained problems in Chapter 4 and then extend them to constrained problems in Chapter 5. The gradients provide much richer information about the function behavior, which the optimizer can use to converge to the optimum more efficiently. In addition, the gradients are used to establish whether the optimizer



**Figure 1.19:** Optimization algorithms can be classified by using the attributes on the rightmost column. As with the problem classification, these attributes are independent, and any combination is possible.

converged to a point that satisfies mathematical optimality conditions, something that is difficult to verify in a rigorous way without gradients. Gradient-based algorithms also include algorithms that use second-order information (curvature). Curvature is even richer information that tells us the rate of the change in the gradient, which provides an idea of where the function will flatten out.

There is a distinction between the order of information provided by the user and the order of information that is actually used in the algorithm. For example, a user might only provide function values to a gradient-based algorithm and rely on the algorithm to internally estimate gradients by requesting additional function evaluations and using finite differences.

In theory, gradient-based algorithms require the functions to be sufficiently smooth (at least  $C^1$  continuous). However, in practice, they can tolerate the occasional discontinuity, as long as this discontinuity does not happen to be at the optimum point.

We devote a considerable portion of this book to gradient-based algorithms because they generally scale better to problems with many design variables, and they have rigorous mathematical criteria for optimality. We also cover the various approaches for computing gradients in detail because the accurate and efficient computation of

these gradients is crucial for the efficacy and efficiency of these methods (Chapter 6).

Current state-of-the-art optimization algorithms also use second-order information to implement Newton-type methods for second-order convergence. However, these algorithms tend to build second-order information based on the provided gradients, as opposed to requiring users to provide the second-order information directly.

Because gradient-based methods require accurate gradients and smooth enough functions, they require more knowledge about the models and optimization algorithm than gradient-free methods. Chapters 3 through 6 are devoted to making the power of gradient-based methods more accessible by providing the necessary theoretical and practical knowledge.

#### 1.4.2 Local Versus Global Search

The many ways to search the design space can be classified as being local or global. Local search takes a series of steps starting from a single point to form a trail of points that hopefully converges to a local optimum. In spite of the name, local methods can traverse large portions of the design space and can even step between convex regions (although this happens by chance). Global search tries to span the whole design space in the hopes of finding the global optimum. As previously mentioned when discussing multimodality, even when using a global method, we cannot prove that any optimum found is a global one except for particular cases.

The local versus global search classification often gets conflated with the gradient-based versus gradient-free attributes because gradient-based methods usually perform a local search. However, these should be viewed as independent attributes because it is possible to use a global search strategy to provide starting points for a gradient-based algorithm. Similarly, some gradient-free algorithms are based on local search strategies.

The choice of search type is intrinsically linked to the modality of the design space. If the design space is unimodal, then a local search will be sufficient, and it will converge to the global optimum. If the design space is multimodal, a local search will converge to an optimum that might be local (or global if we are lucky enough). A global search will increase the likelihood that we converge to a global optimum, but this is by no means guaranteed.

### 1.4.3 Mathematical Versus Heuristic

There is a big divide in how much of an algorithm's iterative process and optimality criteria are based on provable mathematical principles versus heuristics. The iterative process determines the sequence of points that get evaluated in seeking the optimum, while the optimality criteria determine when this iterative process ends. Heuristics are rules of thumb or common sense arguments that are not based on a strict mathematical rationale.

Gradient-based algorithms are usually based on mathematical principles both for the iterative process and for the optimality criteria. Gradient-free algorithms are more evenly split between the mathematical and heuristic for both optimality criteria and iterative procedure. The mathematical ones are often classified *derivative-free optimization* algorithms. Heuristic gradient-free algorithms include a wide variety of nature-inspired algorithms.

Heuristic optimality criteria are an issue because, strictly speaking, they do not prove a given point is a local (let alone global) optimum; they are only expected to find a point that is "close enough". This contrasts with mathematical optimality criteria, which are unambiguous about (local) optimality and converge to the optimum within the limits of the working precision. The mathematical criteria usually require the gradients of the objective and constraints. This is not to suggest that heuristic methods are not useful. Finding a better solution is often desirable regardless of whether or not it is strictly optimal.

Iterative processes based on mathematical principles tend to be more efficient than those based on heuristics. However, some heuristic methods are more robust because they tend to make fewer assumptions about the modality and smoothness of the functions and handle noisy functions more effectively.

Algorithms often mix mathematical arguments and heuristics to some degree. Most mathematical algorithms include constants whose values end up being tuned based on experience. Conversely, algorithms primarily based on heuristics sometimes include steps with mathematical justification.

### 1.4.4 Function Evaluation

The optimization problem setup that we described above assumes that the function evaluations are obtained by solving numerical models of the system. We call these *direct* function evaluations. However, it is possible to create a *surrogate model* (also known as a metamodel) of these models and use them in the optimization process. These surrogate

models can be interpolation-based or projection-based. Surrogate-based optimization is discussed in Chapter 10.

#### 1.4.5 Stochasticity

This attribute is independent of the stochasticity of the model that we mentioned previously, and it is strictly related to whether the optimization algorithm itself contains steps that are determined at random or not.

A deterministic optimization algorithm always evaluates the same points and converge to the same result given the same initial conditions. In contrast, a stochastic optimization algorithm evaluates a different set of points if run multiple times from the same initial conditions, even if the models for the objective and constraints are deterministic. For example, most evolutionary algorithms include steps determined by generating random numbers. Gradient-based algorithms are usually deterministic, but some exceptions exist, such as stochastic gradient descent from the machine learning community.<sup>††</sup>

<sup>††</sup>Stochastic gradient descent is briefly discussed in Chapter 10.

#### 1.4.6 Time Dependence

In this book, we assume that the optimization problem is *static*, where this means that we can solve the complete numerical model at each optimization iteration. For some problems that involve time dependence, we can perform the time integration to solve for the full-time history of the states and then compute the objective and constraint function values for an optimization iteration. This means that every optimization iteration requires solving for the complete time history. An example of this type of problem is a trajectory optimization problem where the design variables are the coordinates representing the path, and the objective is to minimize the total energy expended to get to a given destination. Although such a problem involves a time dependence, we solve a single optimization problem, so we still classify such a problem as static.

For another class of time-dependent optimization problems, however, we solve for a sequence or a time history of decisions at different time instances because we must make decisions as time progresses. These are called *dynamic optimization problems* (also known as dynamic programming or optimal control).<sup>#</sup> In such problems, the design variables are the sequence of decisions, and the decision at a given time instance is influenced by the decisions made in the previous time instances. An example of a dynamic optimization problem would be to optimize the throttle, braking, and steering of a car at each time

<sup>#</sup>There are numerous textbooks on this topic.<sup>1, 2</sup>

1. Bryson et al., *Applied Optimal Control; Optimization, Estimation, and Control*. 1969

2. Bertsekas, *Dynamic programming and optimal control*. 1995

instance such that the overall time in a racecourse is minimized. This is an example of an *optimal control problem*, a type of dynamic optimization problem where a control law is optimized for a dynamical system over a period of time. Dynamic optimization is not broadly covered in this book, except in the context of discrete optimization Section 8.5. Different approaches are used in general, but many of the concepts covered here are instrumental in the numerical solution of optimal control problems.

## 1.5 Selecting an Optimization Approach

This section provides guidance on how to select an appropriate approach for solving a given optimization problem. This process cannot always be boiled down to a simple decision tree; however, it is still helpful to have a framework from which to start. Many of these decisions will become more apparent as you progress through the book and gain experience, so you may want to revisit this section periodically. Eventually, selecting an appropriate methodology will become second nature.

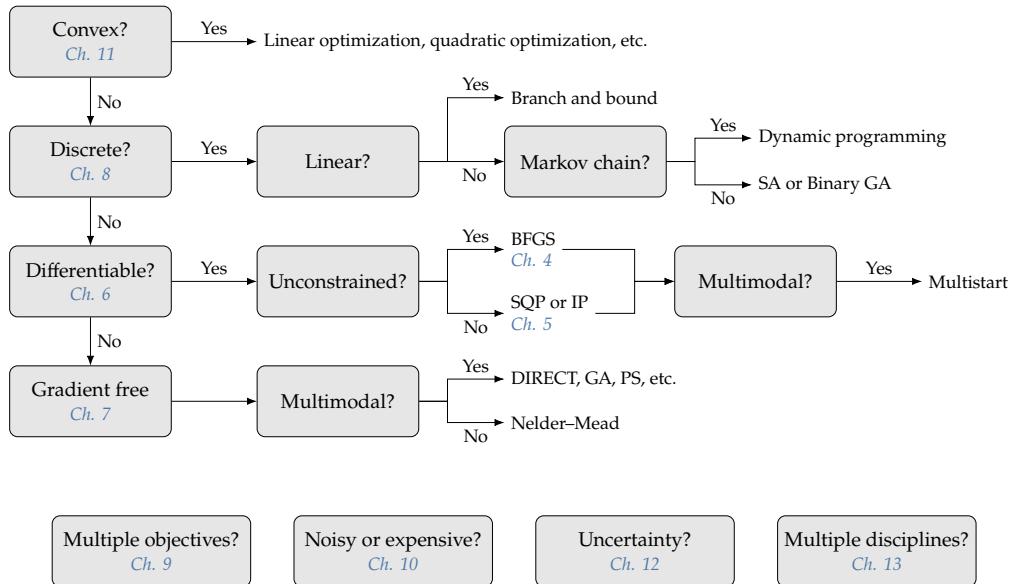


Figure 1.20 outlines one approach to algorithm selection and simultaneously serves as an overview of the chapters in this book. These first two characteristics (convex problem and discrete variables) are not the most common within the broad spectrum of engineering opti-

**Figure 1.20:** Decision tree for selecting an optimization algorithm.

mization problems, but they are the more restrictive in terms of usable optimization algorithms, so we list them first.

The first node asks about convexity. While it is often not immediately apparent if the problem is convex, with some experience, one can usually discern whether attempting to reformulate in a convex manner is likely to be possible. In most instances, convexity occurs for problems with simple objectives and constraints (e.g., linear or quadratic), such as in control applications where the optimization is performed repeatedly. A convex problem can be solved with the more general gradient-based or gradient-free algorithms, but it would be inefficient not to take advantage of the convex formulation structure if we can do so.

The next node asks about discrete variables. Problems with discrete variables are much harder to solve, so we often consider techniques to avoid using discrete variables whenever possible. For example, a wind turbine position in a field could be posed as a discrete variable within a discrete set of options. Alternatively, we could represent the wind turbine position as a continuous variable with two continuous coordinate variables. That level of flexibility may or may not be desirable but will almost always lead to better solutions.

Next, we consider if the model is differentiable or if it can be made differentiable through model improvements. If the problem is high dimensional (more than a few tens of variables as a rule of thumb), gradient-free methods are generally intractable. We would either need to either make the model differentiable or reduce the dimensionality of the problem. Another alternative if the problem is not readily differentiable is to consider surrogate-based optimization (the box labeled “noisy or expensive”). If we go the surrogate-based optimization route, we could still use a gradient-based approach to optimize the surrogate model because most of them are differentiable. Finally, for problems with a relatively small number of design variables, gradient-free methods can be a good fit. The largest variety of algorithms are gradient-free methods, and a combination of experience and testing is needed to determine an appropriate algorithm for the problem at hand.

The bottom rows list additional considerations that fit within most of the algorithms: multiple objectives, surrogate-based optimization for noisy (non-differentiable) or computationally expensive functions, optimizing under uncertainty in the design variables and other model parameters, and MDO architectures.

## 1.6 Notation

By default, a vector  $u$  is a column vector, and thus  $u^T$  is a row vector. We do not use bold vectors or matrices as they are ubiquitous throughout the text. Instead, we follow the convention of many optimization books, which use Greek symbols (such as  $\alpha$  and  $\beta$ ) to represent scalars, lowercase roman letters (such as  $x$  and  $u$ ) for vectors, and capitalized roman letters (such as  $A$  and  $H$ ) as matrices. One of the few exceptions is  $f$ , which is used for scalar functions because that is common usage and most objectives are scalar. Because of the wide variety of topics covered in this book and a desire not to deviate from standard conventions used in some fields, occasional exceptions exist, but these are explicitly noted.

A subscript index,  $x_i$ , refers to the  $i^{\text{th}}$  element in vector  $x$ . Similarly,  $A_{ij}$  is the element at row  $i$  column  $j$  in matrix  $A$ . A superscript index in parenthesis refers to an iteration number, so  $x^{(k)}$ , is the complete vector  $x$  at iteration  $k$ . A superscript star ( $x^*$ ) refers to an optimized quantity.

## 1.7 Summary

Optimization is compelling, and there are opportunities to apply it everywhere. Numerical optimization fully automates the design process but requires expertise in the formulation of the problem, selecting the appropriate optimization algorithm, and using that algorithm. Finally, design expertise is also required to interpret and critically evaluate the optimum results.

There is no single optimization algorithm that is effective in the solution of all types of problems. It is crucial to classify the optimization problem and understand the optimization algorithms' characteristics to select the appropriate algorithm to solve the problem.

## Problems

1.1 Answer *true* or *false* and justify your answer.

- a) MDO arose from the need to consider multiple design objectives.
- b) The preliminary design phase takes place after the conceptual design phase.
- c) Design optimization is a completely automated process from which designers can expect to get their final design.

- d) The design variables for a problem consist of all the inputs needed to compute the objective and constraint functions.
- e) The design variables must always be independent of each other.
- f) An optimization algorithm that is designed for minimization can be used to maximize an objective function without modifying the algorithm.
- g) Compared to the global optimum of a given problem, adding more design variables to that problem results in a global optimum that is no worse than that of the original problem.
- h) Compared to the global optimum of a given problem, adding more constraints sometimes results in a better global optimum.
- i) A function is  $C^1$  continuous if its derivative varies continuously.
- j) All unimodal functions are convex.
- k) Global search algorithms always converge to the global optimum.
- l) Gradient-based methods are based on mathematical principles as opposed to heuristics.
- m) Solving a problem that involves a stochastic model requires a stochastic optimization algorithm.
- n) If a problem is multimodal, you should use a gradient-free optimization algorithm.

1.2 *Plotting a one-dimensional function.* Consider the one-dimensional function

$$f(x) = \frac{1}{12}x^4 + x^3 - 16x^2 + 4x + 12.$$

Plot the function and find the approximate location and classify the minimum point(s). Exploration: Plot other functions to get an intuition about their trends and minima. You can start with simple low-order polynomials and then add higher-order terms, trying different coefficients. Then you can also try non-algebraic functions.

1.3 *Plotting a two-dimensional function.* Consider the two-dimensional function

$$f(x_1, x_2) = x_1^3 + 2x_1x_2^2 - x_2^3 - 20x_1.$$

Plot the function contours and find the approximate location and classify the minimum point(s). Exploration: Similarly to

the suggested exploration in the previous exercise, try plotting different two-dimensional functions to get an intuition about the function trends and minima.

- 1.4 Convert the following problem to the standard formulation (1.4):

$$\begin{aligned} & \text{maximize} && 2x_1^2 - x_1^4 x_2^2 - e^{x_3} + e^{-x_3} + 12 \\ & \text{by varying} && x_1, x_2, x_3 \\ & \text{subject to} && x_1 \geq 1 \\ & && x_2 + x_3 \geq 10 \\ & && x_1^2 + 3x_2^2 \leq 4 \end{aligned} \tag{1.5}$$

- 1.5 *Using an unconstrained optimizer.* Consider the two-dimensional function

$$f(x_1, x_2) = (1 - x_1)^2 + (1 - x_2)^2 + \frac{1}{2} (2x_2 - x_1^2)^2,$$

Plot the contours of this function and find the minimum graphically. Then, use optimization software to find the minimum (see Tip 1.5). Verify that the optimizer converges to the minimum you found graphically. Exploration: 1. Try minimizing the function from Prob. 1.3 starting from different points. 2. Minimize other functions of your choosing. 3. Study the options provided by the optimization software and explore different settings.

- 1.6 *Using a constrained optimizer.* Now we add constraints to Prob. 1.5. The objective is the same, but we now have two inequality constraints:

$$\begin{aligned} & x_1^2 + x_2^2 \leq 1, \\ & x_1 - 3x_2 + \frac{1}{2} \geq 0, \end{aligned}$$

and bound constraints:

$$x_1 \geq 0, \quad x_2 \geq 0.$$

Plot the constraints and identify the feasible region. Find the constrained minimum graphically. Use optimization software to solve the constrained minimization problem. Which of the inequality constraints and bounds are active at the solution?

- 1.7 *Paper review.* Select a paper on design optimization that seems interesting to you, preferably from a peer-reviewed journal. Write the full optimization problem statement in the standard form (1.4)

for the problem solved in the paper. Classify the problem according to Fig. 1.12 and the optimization algorithm according to Fig. 1.19. Use the decision tree in Fig. 1.20 to determine if the optimization algorithm was chosen appropriately. Write a critique of the paper highlighting its strengths and weaknesses.

- 1.8 *Problem formulation.* Choose an engineering system that you are familiar with, and use the process outlined in Fig. 1.4 to formulate a problem for the design optimization of that system. Write the statement in the standard form (1.4). Critique your statement by asking the following: Does the objective function truly capture the design intent? Are there other objectives that could be considered? Do the design variables provide enough freedom? Are the design variables bounded such that non-physical designs are prevented? Are you sure you have included all the constraints needed to get a practical design? Can you think of any loophole that the optimizer can exploit in your statement?

## A Short History of Optimization

# 2

This chapter provides helpful historical context for algorithms discussed in this book. Nothing else in the book depends on familiarity with this chapter, so it can be skipped. However, this history makes connections between the various topics that will enrich the big picture of optimization as you become familiar with the material in the rest of the book. Optimization has a long history that started with geometry problems solved by ancient Greek mathematicians. The invention of algebra and calculus opened the door to many more problems, and the advent of numerical computing increased the range of problems that could be solved in terms of type and scale.

By the end of this chapter you should be able to:

1. Appreciate a range of historical advances in optimization.
2. Describe some current frontiers in optimization.

### 2.1 The First Problems: Optimizing Length and Area

Ancient Greek and Egyptian mathematicians made numerous contributions to geometry, including solving optimization problems that involved lengths and areas. They adopted a geometric approach to solving problems that are now generally more easily solved using calculus.

Archimedes of Syracuse (287–212 BCE) showed that of all possible spherical caps of a given surface area, hemispherical caps have the largest volume. Euclid of Alexandria (325–265 BCE) showed that the shortest distance between a point and a line is the segment perpendicular to that line. He also proved that among rectangles of a given perimeter, the square has the largest area.

Geometric problems involving perimeter and area were of actual practical value. The classic example of such practicality is Dido's problem. According to the legend, Queen Dido, who had fled to Tunis,

purchased from a local leader as much land as could be enclosed by an ox's hide. The leader agreed because this seemed like a modest amount of land. To maximize her land area, queen Dido had the hide cut into narrow strips to make the longest possible string. Then, she intuitively found the curve that maximizes the area enclosed by string: a semicircle with the diameter segment set along the sea coast. As a result of the maximization, she acquired enough land to found the ancient city of Carthage. Later, Zenodorus (200–140 BCE) proved this optimal solution using geometrical arguments. A rigorous solution to this problem requires using calculus of variations, which was invented much later (see Section 2.2).

Geometric optimization problems are also applicable to laws of physics. Hero of Alexandria (10–70 CE) derived the law of reflection by finding the shortest path for light reflecting from a mirror, which results in an angle of reflection equal to the angle of incidence (Fig. 2.2).

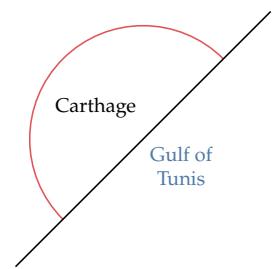
## 2.2 Optimization Revolution: Derivatives and Calculus

The scientific revolution generated significant optimization developments in the 17th and 18th centuries that intertwined with other mathematics and physics developments.

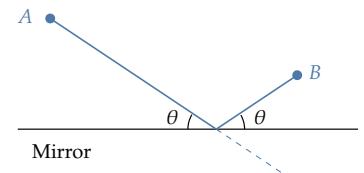
In the early 17th century, Johannes Kepler published a book in which he derived the optimal dimensions of a wine barrel.<sup>3</sup> He became interested in this problem when he bought a barrel of wine, and the merchant charged him based on a diagonal length (see Fig. 2.3). This outraged Kepler because he realized that the amount of wine could vary for the same diagonal length, depending on the barrel proportions.

Incidentally, Kepler also formulated an optimization problem when looking for his second wife, seeking to maximize the likelihood of satisfaction. This “marriage problem” later became known at the “secretary problem”, which is an optimal stopping problem that has since been solved using dynamic optimization (mentioned in Section 1.4.6 and discussed in Section 8.5).<sup>4</sup>

Willebrord Snell discovered the law of refraction in 1621, a formula that describes the relationship between the angles of incidence and refraction when light passes through a boundary between two different media such as air, glass, or water. While Hero solved a length minimization problem to derive the law of reflection, Snell minimized time. These laws were generalized by Fermat in the *principle of least time* (or Fermat’s principle), which states that a ray of light going from one point to another follows the path that takes the least time.

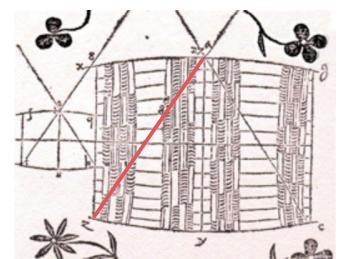


**Figure 2.1:** Queen Dido intuitively maximized the area for a given perimeter to found the city of Carthage.



**Figure 2.2:** The law of reflection can be derived by minimizing the length of the beam of light.

3. Kepler, *Nova stereometria doliorum vinariorum* (*New solid geometry of wine barrels*). 1615



**Figure 2.3:** Wine barrels were measured by inserting a ruler in the taphole until it hit the corner.

4. Ferguson, *Who Solved the Secretary Problem?* 1989

Pierre de Fermat derived Snell's law by applying the principle of least time, and in the process, he devised a mathematical technique for finding maxima and minima using what amounted to derivatives (he missed out on generalizing the notion of derivative, which came later in the development of calculus).<sup>5</sup>

Isaac Newton wrote about a numerical technique in 1669 to find the roots of polynomials by successively linearizing them, achieving quadratic convergence. In 1687, he used this technique to find the roots of a non-polynomial equation (Kepler's equation),\* but only after using polynomial expansions. In 1690, Joseph Raphson improved on Newton's method by keeping all the decimals in each linearization and made it a fully iterative scheme. The multivariable "Newton's method" that is widely used today was actually introduced in 1740 by Thomas Simpson. He generalized the method by using the derivatives (which allowed for solving non-polynomial equations without expansions) and by extending it to a system of two equations and two unknowns.

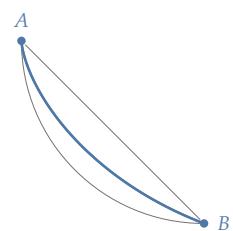
In 1685, Newton studied a shape optimization problem where he sought the shape of a body of revolution that minimizes fluid drag and even mentioned a possible application to ship design. Although he used the wrong model for computing the drag, he correctly solved what amounted to a calculus of variations problem.

In 1696, Johann Bernoulli challenged other mathematicians to find the path of a body subject to gravity that minimizes the travel time between two points of different heights. This is now a classic calculus of variations problem called the brachistochrone problem (Fig. 2.4). Bernoulli already had a solution that he kept secret. Five mathematicians respond with solutions: Newton, Jakob Bernoulli (Johann's brother), Gottfried Wilhelm von Leibniz, Ehrenfried Walther von Tschirnhaus, and Guillaume de l'Hôpital. Newton reportedly started working on the problem as soon as he received it and stayed up all night before sending the solution anonymously to Bernoulli the next day.

Starting in 1736, Leonhard Euler derived the general optimality conditions for solving calculus of variations problems, but the derivation included geometric arguments. In 1755, Joseph-Louis Lagrange used a purely analytic approach to derive the same optimality conditions (he was 19 years old at the time!). Euler recognized Lagrange's derivation, which uses variations of a function, as a superior approach and adopted it, calling it "calculus of variations". This is a second-order partial differential equation that has become known as the Euler–Lagrange equation. Lagrange uses the Euler–Lagrange equation to develop a reformulation of classical mechanics in 1788, which became known

5. Fermat, *Methodus ad disquirendam maximam et minimum (Method for the study of maxima and minima)*. 1636

\*Kepler's equation describes orbits by  $E - e \sin(E) = M$ , where  $M$  is the mean anomaly,  $e$  is the eccentricity, and  $E$  is the eccentric anomaly. This equation does not have a closed-form solution for  $E$ .



**Figure 2.4:** Suppose that you have a bead on a wire that goes from  $A$  to  $B$ . The brachistochrone curve is the shape of the wire that minimizes the time for the bead to slide between the two points under gravity alone. It is faster than a straight line trajectory or a circular arc.

as Lagrangian mechanics. When deriving the general equations of equilibrium for problems with constraints, Lagrange introduces the “method of the multipliers”.<sup>6</sup> Lagrange multipliers eventually become a fundamental concept in constrained optimization (Chapter 5).

In 1784, Gaspard Monge developed a geometric method to solve a transportation problem. Although the method is not entirely correct, it marks the establishment of combinatorial optimization, a branch of discrete optimization (Chapter 8).

<sup>6</sup>. Lagrange, *Mécanique analytique*. 1788

## 2.3 The Birth of Optimization Algorithms

There were several more theoretical contributions related to optimization in the 19th century and the early 20th century. However, it was not until the 1940s that optimization started to gain traction with the development of algorithms and their use in practical applications, thanks to the advent of computer hardware.

In 1805, Adrien-Marie Legendre described the method of least squares, which was used to predict asteroid orbits and curve fitting. Frederick Gauss published a rigorous mathematical foundation for the method of least squares and claims he used it to predict the orbit of the asteroid Ceres in 1801. Legendre and Gauss engaged in a bitter dispute on who first developed the method.

In one of his 789 papers, Augustin-Louis Cauchy proposed the steepest descent method for solving systems of nonlinear equations.<sup>7</sup> He does not seem to put much thought into it and promises a “paper to follow” on the subject that never happens. He proposed this method for solving systems of nonlinear equations, but it is directly applicable to unconstrained optimization (Chapter 4).

<sup>7</sup>. Cauchy, *Méthode générale pour la résolution des systèmes d'équations simultanées*. 1847

In 1902, Gyula Farkas proved a theorem on the solvability of a system of linear inequalities. This became known as Farkas’ lemma, which is crucial in the derivation of the Karush–Kuhn–Tucker optimality conditions for constrained problems (see below and Chapter 5).

In 1917, Harris Hancock publishes the first textbook on optimization, which includes the optimality conditions for multivariable unconstrained and constrained problems.<sup>8</sup>

<sup>8</sup>. Hancock, *Theory of Minima and Maxima*. 1917

In 1932, Karl Menger presented “the messenger problem”,<sup>9</sup> an optimization problem that seeks to minimize the shortest travel path that connects a set of destinations, observing that going to the closest point each time does not in general result in the shortest overall path. This is a combinatorial optimization problem that later becomes known as the traveling salesman problem, one of the most intensively studied problems in optimization (Chapter 8).

<sup>9</sup>. Menger, *Das botenproblem*. 1932

In 1939, William Karush derived the necessary conditions for inequality constrained problems in his master’s thesis. This generalizes the method of Lagrange multipliers, which only allowed for equality constraints. Harold Kuhn and Albert Tucker independently rediscover these conditions and publish their seminal paper in 1951.<sup>10</sup> These became known as the Karush–Kuhn–Tucker (KKT) conditions, which constitute the foundation of gradient-based constrained optimization algorithms (Section 5.2).

Leonid Kantorovich developed a technique to solve linear programming problems in 1939 after having been given the task of optimizing production in the Soviet government plywood industry. However, his contribution was neglected for ideological reasons. In the United States, Tjalling Koopmans rediscovers linear programming in the early 1940s when working on ship transportation problems. In 1947, George Dantzig published the first complete algorithm to solve linear programming problems—the simplex algorithm. In the same year, von Neumann develops the theory of duality for linear programming problems. Kantorovich and Koopmans later shared the 1975 Nobel Memorial Prize in Economic Sciences “for their contributions to the theory of optimum allocation of resources”. Dantzig was not included, presumably because his work was not as applied. The development of the simplex algorithm and the widespread practical applications of linear programming spark a revolution in optimization. The first international conference on optimization, the International Symposium on Mathematical Programming, was held in Chicago in 1949.

In 1951, George Box and Kenneth Wilson developed the response surface methodology (surrogate modeling), which enables optimization of systems based on experimental data (as opposed to a physics-based model). They developed a method to build a quadratic model where the number of data points scales linearly with the number of inputs, instead of exponentially, striking a balance between accuracy and ease of application. In the same year, Danie Krige develops a surrogate model based on a stochastic process, which is now known as “kriging”. He developed this model in his master’s thesis to estimate the most likely distribution of gold based on a limited number of borehole samples.<sup>11</sup> These approaches are foundational in surrogate-based optimization (Chapter 10).

In 1952, Harry Markowitz published a paper on portfolio theory that formalizes the idea of investment diversification, marking the birth of modern financial economics.<sup>12</sup> The theory is based on a quadratic optimization problem. He received the 1990 Nobel Memorial Prize in Economic Sciences.

<sup>10</sup>. Karush, *Minima of Functions of Several Variables with Inequalities as Side Constraints*. 1939

<sup>11</sup>. Krige, *A statistical approach to some mine valuation and allied problems on the Witwatersrand*. 1951

<sup>12</sup>. Markowitz, *Portfolio selection*. 1952

In 1955, Lester Ford and Delbert Fulkerson created the first known algorithm to solve the maximum flow problem, which has applications in transportation, electrical circuits, and data transmission. While the problem could already be solved with the simplex algorithm, they proposed a more efficient algorithm for this specialized problem.

In 1957, Richard Bellman derived the necessary optimality conditions for dynamic programming problems. These are expressed in what became known as the Bellman equation (Section 8.5), which was first applied to engineering control theory, and subsequently became a core principle in the development of economic theory.

In 1959, William Davidon developed the first quasi-Newton method to solve nonlinear optimization problems that rely on approximations of the curvature based on gradient information. He was motivated by his work at Argonne National Lab, where he used a coordinate descent method to perform an optimization that kept crashing the computer before converging. Although Davidon's approach was a breakthrough in nonlinear optimization, his original paper was rejected. It was eventually published more than 30 years later in the first issue of the SIAM Journal of Optimization.<sup>13</sup> Fortunately, his valuable insight had been recognized well before that by Roger Fletcher and Michael Powell, who developed the method further.<sup>14</sup> The method became known as DFP (Section 4.4.4).

Another quasi-Newton approximation method was independently proposed in 1970 by Charles Broyden, Roger Fletcher, Donald Goldfarb, and David Shanno, now called the BFGS approximation (Fig. 2.5). Larry Armijo, A. Goldstein, and Philip Wolfe develop the conditions for the line search in gradient-based methods that ensure convergence (see Section 4.3.2).<sup>15</sup>

With these developments in unconstrained optimization, researchers seek methods to solve constrained problems as well. Penalty and barrier methods are developed but fall out of favor because of numerical issues (see Section 5.3).

In another effort to solve nonlinear constrained problems, Robert Wilson proposed the sequential quadratic programming (SQP) method in his Ph.D. thesis.<sup>16</sup> SQP essentially consists in applying the Newton method to solve the KKT conditions (see 5.4). Shih-Ping Han re-invented SQP in 1976<sup>17</sup> and Michael Powell popularized this method in a series of papers starting from 1977.<sup>18</sup>

There were attempts to model the natural process of evolution starting in the 1950s. In 1975, John Holland proposed genetic algorithms (GA) to solve optimization problems (Section 7.5).<sup>19</sup> Research in GAs increased dramatically after that, thanks in part to the exponential

<sup>13</sup>. Davidon, *Variable Metric Method for Minimization*. 1991

<sup>14</sup>. Fletcher et al., *A Rapidly Convergent Descent Method for Minimization*. 1963



**Figure 2.5:** Broyden, Fletcher, Goldfarb, and Shanno (BFGS) at the NATO Optimization Meeting (Cambridge, UK, 1983), a seminal meeting for nonlinear optimization (courtesy of Prof. John Dennis).

<sup>15</sup>. Wolfe, *Convergence Conditions for Ascent Methods*. 1969

<sup>16</sup>. Wilson, *A simplicial algorithm for concave programming*. 1963

<sup>17</sup>. Han, *Superlinearly convergent variable metric algorithms for general nonlinear programming problems*. 1976

<sup>18</sup>. Powell, *Algorithms for nonlinear constraints that use Lagrangian functions*. 1978

<sup>19</sup>. Holland, *Aptation in Natural and Artificial Systems*. 1975

increase in computing power (Section 7.5).

Hooke *et al.*<sup>20</sup> proposed a gradient-free method, which they call “pattern search.” In 1965, Nelder *et al.*<sup>21</sup> developed the nonlinear simplex method another gradient-free nonlinear optimization based on heuristics (Section 7.3). (This has no connection to the simplex algorithm for linear programming problems mentioned above.)

The Mathematical Programming Society was founded in 1973, an international association for researchers active in optimization. It was eventually renamed Mathematical Optimization Society in 2010.

Narendra Karmarkar presented a revolutionary new method in 1984 to solve large-scale LPs as much as a hundred times faster than the simplex method.<sup>22</sup> The New York Times publishes a news item on the first page with the headline “Breakthrough in Problem Solving.” This starts the age of interior point methods, which are related to the barrier methods dismissed in the 1960s. Interior point methods are eventually adapted to solve nonlinear problems (see Section 5.5) and contribute to the unification of linear and nonlinear optimization.

## 2.4 The Last Decades

The relentless exponential increase in computer power through the 1980s and beyond has made it possible to perform engineering design optimization with increasingly sophisticated models, including multidisciplinary models. The increased computer power has also been contributing to the gain in popularity of heuristic optimization algorithms. Computer power has also increased the use of neural networks and the explosive rise of artificial intelligence.

The field of optimal control flourished after Bellman’s contribution to dynamic programming. Another important optimality principle for control, the maximum principle, was derived by Pontryagin *et al.*<sup>23</sup>. This principle makes it possible to transform a calculus of variations problem into a nonlinear optimization problem. Gradient-based nonlinear optimization algorithms were then used to numerically solve for optimal trajectories of rockets and aircraft, with an adjoint method to compute the gradients of the objective with respect to the control histories.<sup>24</sup> The adjoint method efficiently computes gradients with respect to large numbers of variables and proved to be useful in other disciplines. Optimal control expands to include the optimization of feedback control laws that guarantee closed-loop stability. Optimal control approaches include model-predictive control, which is widely used today.

In 1960, Schmit<sup>25</sup> proposed coupling numerical optimization with structural computational models to perform structural design, establish-

<sup>20</sup>. Hooke *et al.*, “Direct Search” Solution of Numerical and Statistical Problems. 1961

<sup>21</sup>. Nelder *et al.*, A Simplex Method for Function Minimization. 1965

<sup>22</sup>. Karmarkar, *A New Polynomial-Time Algorithm for Linear Programming*. 1984

<sup>23</sup>. Pontryagin *et al.*, *The Mathematical Theory of Optimal Processes*. 1961

<sup>24</sup>. Bryson Jr, *Optimal Control—1950 to 1985*. 1996

<sup>25</sup>. Schmit, *Structural Design by Systematic Synthesis*. 1960

ing the field of structural optimization. Five years later, he presented applications, including aerodynamics and structures, which represents the first known MDO application.<sup>26</sup> The direct method for computing gradients for structural computational models is developed shortly after that,<sup>27</sup> eventually followed by the adjoint method (Section 6.7).<sup>28</sup> In this early work, the design variables were the cross-sectional areas of the members of a truss structure. Researchers then added joint positions in the set of design variables. Structural optimization was generalized further with shape optimization, which optimized the shape of arbitrary three-dimensional structural parts.<sup>29</sup> Another significant development was topology optimization, where a structural layout emerges from a solid block of material.<sup>30</sup> It took many years of further development in algorithms and computer hardware for structural optimization to be widely adopted by industry, but now this capability has made its way to commercial software.

Aerodynamic shape optimization began when Pironneau<sup>31</sup> used optimal control techniques to minimize the drag of a body by varying its shape (the “control” variables). Jameson<sup>32</sup> extended the adjoint method with more sophisticated computational fluid dynamics models and applied it to aircraft wing design. CFD-based optimization applications spread beyond aircraft wing design to the shape optimization of wind turbines, hydrofoils, ship hulls, and automobiles. The adjoint method was then generalized for any discretized system of equations (Section 6.7).

MDO developed rapidly in the 1980s, following the application of numerical optimization techniques to structural design. The first conference in MDO, the Multidisciplinary Analysis and Optimization Conference, took place in 1985. The first MDO applications focused on coupling the aerodynamics and structures in wing design, and other early applications integrated structures and controls.<sup>33</sup> The development of MDO methods progressed efforts towards decomposing the problem into optimization subproblems, leading to distributed MDO architectures.<sup>34</sup> Sobiesczanski–Sobieski<sup>35</sup> proposed a formulation for computing the derivatives for coupled systems, which is needed when performing MDO with gradient-based optimizers. This concept was later combined with the adjoint method to compute coupled derivatives efficiently.<sup>36</sup> More recently, efficient coupled derivative computation and hierarchical solvers have made it possible to solve large-scale MDO problems<sup>37</sup> (see Chapter 13). Engineering design has been focusing on achieving improvements made possible by considering the interaction of all relevant disciplines. MDO applications have extended beyond aircraft to the design of bridges, buildings, automobiles, ships, wind

26. Schmit *et al.*, *Synthesis of an Airfoil at Supersonic Mach Number*. 1965

27. Fox, *Constraint Surface Normals for Structural Synthesis Techniques*. 1965

28. Arora *et al.*, *Methods of Design Sensitivity Analysis in Structural Optimization*. 1979

29. Haftka *et al.*, *Structural shape optimization—A survey*. 1986

30. Eschenauer *et al.*, *Topology optimization of continuum structures: A review*. 2001

31. Pironneau, *On optimum design in fluid mechanics*. 1974

32. Jameson, *Aerodynamic Design via Control Theory*. 1988

33. Sobiesczanski–Sobieski *et al.*, *Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments*. 1997

34. Martins *et al.*, *Multidisciplinary Design Optimization: A Survey of Architectures*. 2013

35. Sobiesczanski–Sobieski, *Sensitivity of Complex, Internally Coupled Systems*. 1990

36. Martins *et al.*, *A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design*. 2005

37. Hwang *et al.*, *A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives*. 2018

turbines, and spacecraft.

In continuous nonlinear optimization, SQP has remained state-of-the-art since its popularization in the late 1970s. However, the interior point approach, which, as mentioned previously, revolutionized linear optimization, was successfully adapted to solve nonlinear problems and has made great strides since the 1990s.<sup>38</sup> Today, both SQP and interior point methods are considered to be state-of-the-art.

Interior point methods have contributed to the connection between linear and nonlinear optimization, which were treated as entirely separate fields before 1984. Today, state-of-the-art linear optimization software packages have options for both the simplex and interior-point approaches because the best approach depends on the problem.

Convex optimization emerged as a generalization of linear optimization (see Chapter 11). Like linear optimization, it was initially mostly used in operations research applications,<sup>†</sup> (such as transportation, manufacturing, supply chain management, and revenue management) and there were only a few applications in engineering. Since the 1990s, convex optimization has increasingly been used in engineering applications, including optimal control, signal processing, communications, and circuit design. A disciplined convex programming methodology facilitated this expansion to construct convex problems and convert them to a solvable form.<sup>39</sup> New classes of convex optimization problems have also been developed, such as geometric programming (see Section 11.6), semidefinite programming, and second-order cone programming.

As mathematical models became increasingly complex computer programs and given the need to differentiate those models when performing gradient-based optimization, new methods have been developed to compute derivatives. Wengert<sup>40</sup> was among the first to propose the automatic differentiation of computer programs (or algorithmic differentiation). The reverse mode of algorithmic differentiation, which is equivalent to the adjoint method applied to computer programs, was proposed later (see Section 6.6).<sup>41</sup> This field has evolved immensely since then, with techniques to handle more functions and increase efficiency. Algorithmic differentiation tools have been developed for an increasing number of programming languages. One of the more recently developed programming languages, Julia, features prominent support for algorithmic differentiation. At the same time, algorithmic differentiation has now been spread to a wide range of applications.

Another technique to compute derivatives numerically, the complex-step derivative approximation, was proposed by Squire *et al.*<sup>42</sup> Soon after, this technique was generalized to computer programs, applied to computational fluid dynamics, and found to be related to automatic

<sup>38.</sup> Wright, *The interior-point revolution in optimization: history, recent developments, and lasting consequences*. 2005

<sup>†</sup>The field of operations research was established in World War II to help make better strategical decisions.

<sup>39.</sup> Grant *et al.*, *Global Optimization—From Theory to Implementation*. 2006

<sup>40.</sup> Wengert, *A Simple Automatic Derivative Evaluation Program*. 1964

<sup>41.</sup> Speelpenning, *Compiling fast partial derivatives of functions given by algorithms*. 1980

<sup>42.</sup> Squire *et al.*, *Using Complex Variables to Estimate Derivatives of Real Functions*. 1998

differentiation (see Section 6.5).<sup>43</sup>

The pattern search algorithms that Hooke and Jeeves, and Nelder and Meade developed were disparaged by applied mathematicians, who preferred the rigor and efficiency of the gradient-based methods developed soon after that. Nevertheless, they were further developed and remain popular with engineering practitioners because of their simplicity. Pattern search methods experienced a renaissance in the 1990s with the development of convergence proofs that added mathematical rigor and the availability of more powerful parallel computers.<sup>44</sup> Today, pattern search methods remain a useful option (and sometimes the only one) for some types of optimization problems.

Global optimization algorithms also experienced further developments. Jones *et al.*<sup>45</sup> developed the DIRECT algorithm, which uses a rigorous approach to find the global optimum (Section 7.4).

The first genetic algorithms started the development of the broader class of evolutionary optimization algorithms inspired more broadly by natural and societal processes. Optimization by simulated annealing represents one of the early examples of this broader perspective.<sup>46</sup> Another example is particle swarm optimization (PSO) (see Section 7.6).<sup>47</sup> Since then, there has been an explosion in the number of evolutionary algorithms, inspired by any process imaginable (see the side note at the end of Section 7.2 for a partial list). Evolutionary algorithms have remained heuristic and have not experienced the mathematical treatment applied to pattern search methods.

There has been a sustained interest in surrogate-models (also known as metamodels) since the seminal contributions in the 1950s. Kriging surrogate models are still being used and have been the focus of many improvements, but new techniques such as radial-basis functions have also emerged.<sup>48</sup> Surrogate-based optimization is now an area of active research (see Chapter 10).

Artificial intelligence (AI) has experienced a revolution in the last decade and is connected to optimization in several ways. The early AI efforts focused on solving problems that could be described formally, such as design optimization problem statements. Today, AI solves problems that are difficult to describe formally, such as face recognition. This new capability is made possible by the development of deep learning neural networks, the availability of large datasets for training the neural networks, and increased computer power. Deep learning neural networks learn to map a set of inputs to a set of outputs based on training data and can be viewed as a type of surrogate model (see Section 10.5). These networks are trained using optimization algorithms that minimize a loss function (analogous to model error), but they

43. Martins *et al.*, *The Complex-Step Derivative Approximation*. 2003

44. Torczon, *On the Convergence of Pattern Search Algorithms*. 1997

45. Jones *et al.*, *Lipschitzian optimization without the Lipschitz constant*. 1993

46. Kirkpatrick *et al.*, *Optimization by Simulated Annealing*. 1983

47. Kennedy *et al.*, *Particle Swarm Optimization*. 1995

48. Forrester *et al.*, *Recent advances in surrogate-based optimization*. 2009

require specialized optimization algorithms such as stochastic gradient descent.<sup>49</sup> The gradients for this problem are efficiently computed with backpropagation, which is a specialization of the reverse mode of AD.<sup>50</sup>

## 2.5 Summary

This history of optimization is as old as human civilization and has had many twists and turns. Ancient geometric optimization problems that were correctly solved by intuition required mathematical developments that were only realized much later. The discovery of calculus laid out the foundations for optimization. Computer hardware and algorithms then enabled the development and deployment of numerical optimization.

Numerical optimization was first motivated by operations research problems but eventually made its way into engineering design. Soon after numerical models were developed to simulate engineering systems, the idea arose to couple those models to optimization algorithms in an automated cycle to optimize the design of such systems. The first application was in structural design, but many other engineering design applications followed, including applications coupling multiple disciplines, establishing MDO.

Many insightful connections have been made in the history of optimization, and the trend has been to unify the theory and methods. There are no doubt more connections to be made.

49. Bottou *et al.*, *Optimization Methods for Large-Scale Machine Learning*. 2018

50. Baydin *et al.*, *Automatic Differentiation in Machine Learning: a Survey*. 2018

In the introductory chapter, we discussed function characteristics from the point of view of the function's output—the black box view shown in Fig. 1.13. Here, we discuss *how* the function is modeled and computed. The more understanding and access you have to the model, the more you can do to solve the optimization problem more effectively. We explain the errors involved in the modeling process so that we can interpret optimization results correctly.

By the end of this chapter you should be able to:

1. Identify different types of numerical error and understand some of the limitations of finite precision arithmetic.
2. Estimate an algorithm's rate of convergence.
3. Use Newton's method to solve systems of equations.

### 3.1 Model Development for Analysis Versus Optimization

A good understanding of numerical models and solvers is essential as numerical optimization demands more of the models and solvers than does pure analysis. In an analysis, or a parametric study, we may cycle through a range of plausible configurations. However, optimization algorithms seek to explore the solutions space and therefore intermediate evaluations may use atypical input combinations. The mathematical model, numerical model, and solver, need to be robust enough to handle these wide-ranging inputs without artificially constraining the solution space. It is important to explore the behavior of the model across a wide range of inputs before attempting optimization.

A related issue is that an optimizer exploits errors in ways an engineering designer would not do in analysis. For example, consider an aerodynamic model of a car. In a parametric study we might try a dozen designs, compare the drag, and choose the best one. If we passed this procedure to an optimizer it would flatten the car to zero

height—the minimum drag solution. Thus, for optimization we often need to develop additional models, in this case modeling cargo and structural requirements. The parametric designer considered these things implicitly and approximately, but in optimization we need to explicitly model these requirements and pose them as constraints.

Another consideration that affects both the mathematical and the numerical model is the overall computational cost of optimization. An analysis might only be run dozens of times, whereas an optimization often runs the analysis thousands of times. This computational cost can affect the level of fidelity, or discretization we can afford to use.

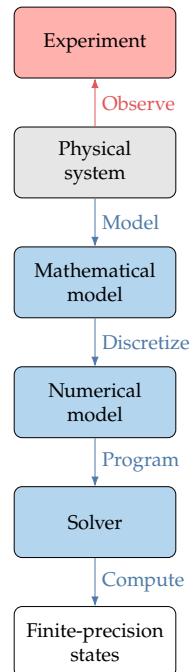
The level of precision that is desirable for analysis is often insufficient for optimization. In an analysis, a few digits of precision may be sufficient. However, using fewer significant digits limits the types of optimization algorithms we can use effectively, convergence failures can cause premature termination of algorithms, and noisy outputs can severely mislead or terminate an optimization prematurely. A common source of these errors involves programs that work through input and output files. Even though the underlying code may use double-precision arithmetic, output files rarely include all the significant digits (another separate issue is that reading and writing files at every iteration is that it considerably slows down the analysis).

Another common source of errors involves converging systems of equations, as discussed later in this chapter. Optimization generally requires tighter tolerances than are used for analysis. Sometimes this is as easy as changing a default tolerance, and other times we need to rethink the solvers we use.

### 3.2 Modeling Process and Types of Errors

Design optimization problems usually involve modeling a physical system so that we can compute the objective and constraint function values for a given design. The steps in the modeling process are shown in Fig. 3.1. The physical system represents the reality that we want to *model*. The mathematical model can range from simple mathematical expressions to sets of continuous differential or integral equations for which closed-form solutions over an arbitrary domain are not possible. When that is the case, we must *discretize* the continuous equations to obtain the numerical model. This numerical model must then be *programmed* using a computer language to develop a numerical solver. Finally, the solver *computes* the system state variables using finite-precision arithmetic.

Each of these steps in the modeling process introduces an error.



**Figure 3.1:** Physical problems are modeled and then solved numerically to produce function values.

*Modeling errors* are introduced in the idealization and approximations performed in the derivation of the mathematical model. The errors involved in the rest of the process are all *numerical errors*, which we detail in Section 3.5. The total error is the sum of the modeling errors and numerical errors.

Validation and verification processes enable us to quantify these errors. Verification is concerned with making sure the errors introduced by the discretization, and the numerical computations are acceptable. In addition, verification aims to make sure there are no bugs in the code that introduce errors. Validation involves comparing the numerical results with experimental observations of the physical system, which are themselves subject to experimental errors. By making these comparisons, we can validate the modeling step of the process and make sure that the idealizations and assumptions in developing the mathematical model are acceptable.

These errors relate directly to the concepts of precision and accuracy. An *accurate* solution is one that compares well with the real physical system (validation), while a *precise* solution just means that the numerical coding and solution are solved correctly (verification).

---

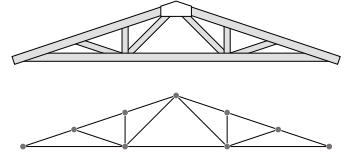
**Example 3.1:** Modeling a structure.

As an example of physical system, consider the timber roof truss structure shown in Fig. 3.2. A typical idealization of such a structure assumes that the wood is an homogeneous material and that the joints are pinned. The loads are applied only at the joints, and the weight of the structure does not contribute to the loading. It is also usual to assume that the displacements are small relative to the dimensions of the truss members. The structure is discretized by pinned bar elements. The discrete governing equations for any truss structure can be derived by enforcing equilibrium at each joint, or more generally, by using the finite-element method. This leads to the linear system,

$$Ku = f,$$

where  $K$  is the stiffness matrix,  $f$  is the vector of applied loads, and  $u$  are the displacements that we want to compute. At each joint, there are two degrees of freedom (horizontal and vertical) that describe the displacement and applied force. Since there are 9 joints, each with 2 degrees of freedom, the size of this linear system is 18.

---



**Figure 3.2:** Timber roof truss and idealized model.

### 3.3 Numerical Models as Residual Equations

Mathematical models vary greatly in complexity and scale. In the simplest case, a model can be represented by one or more explicit

functions, which are easily coded and computed. Many examples in this book use explicit functions for simplicity. In practice, however, most numerical models are defined by *implicit* equations. Implicit equations can be written in the *residual form*,

$$r_i(u_1, \dots, u_n) = 0 \quad i = 1, \dots, n. \quad (3.1)$$

where  $r$  is a vector of residuals that has the same size as the vector of state variables  $u$ . The equation defining the residuals could be any expression that can be coded in a computer program. No matter how complex the mathematical model, it can always be written as a set of equations in this form, which we write more compactly as  $r(u) = 0$ .

This residual notation can still be used to represent explicit functions, so we can use it for all the functions in a model without loss of generality. Suppose we have the explicit function  $u_f \triangleq f(u)$ , where  $u$  is a vector and  $u_f$  is the scalar function value (and not one of the components of  $u$ ). We can rewrite this function as a residual equation by moving all the terms to one side to get a  $r(u_f) = f(u) - u_f = 0$ . Even though it might seem more natural to use explicit functions, we might be motivated to use the residual form to write the whole model in the compact notation,  $r(u) = 0$ . This will be helpful in later chapters when computing derivatives (Chapter 6) and solving systems with multiple components (Chapter 13).

---

**Example 3.2:** Expressing an explicit function as an implicit equation.

Suppose we have the following mathematical model,

$$\begin{aligned} u_1^2 + 2u_2 - 1 &= 0, \\ u_1 + \cos(u_1) - u_2 &= 0, \\ f(u_1, u_2) &= u_1 + u_2. \end{aligned} \quad (3.2)$$

The first two equations are written as implicit functions and the third equation is given as an explicit function. The first equation could be manipulated to obtain an explicit function of either  $u_1$  or  $u_2$ . The second equation does not have a closed-form solution and cannot be written as an explicit function for  $u_1$ . The third equation is an explicit function of  $u_1$  and  $u_2$ . Given these equations, we might decide to solve the first two equations for  $u_1$  and  $u_2$  using a nonlinear solver and then evaluate  $f(u_1, u_2)$ . However, we can write the whole system as implicit residual equations by defining the value of  $f(u_1, u_2)$  as  $u_3$ ,

$$\begin{aligned} r_1(u_1, u_2) &= u_1^2 + 2u_2 - 1 &= 0, \\ r_2(u_1, u_2) &= u_1 + \cos(u_1) - u_2 &= 0, \\ r_3(u_1, u_2, u_3) &= u_1 + u_2 - u_3 &= 0. \end{aligned} \quad (3.3)$$

You can use the same nonlinear solver to solve for all three equations simultaneously.

---

The *governing equations* of a model determine the *state* of a given physical system at specific conditions. Many governing equations consist of differential equations, which require discretization to obtain implicit equations that can be solved numerically (see Section 3.4). After discretization, the governing equations can always be written as  $r(u) = 0$ .

**Example 3.3:** Implicit and explicit equations in structural analysis.

The linear system from Ex. 3.1 can be obtained by a finite-element discretization of the governing equations. This is an example of a set of implicit equations, which we can write as a set of residuals,

$$r(u) = Ku - f = 0, \quad (3.4)$$

where  $u$  are the state variables. While the solution for  $u$  could be written as an explicit function,  $u = K^{-1}f$ , this is usually not done. Instead, we use a linear solver that does not explicitly form the inverse of the stiffness matrix.

In addition to computing the displacements, we might also want to compute the axial stress in each of the 15 truss members. This is an explicit function of the displacements, which is given by the linear relationship

$$\sigma = Su, \quad (3.5)$$

where  $S$  is a  $15 \times 18$  matrix. Another example of an explicit function is the computation of the structural mass, which is

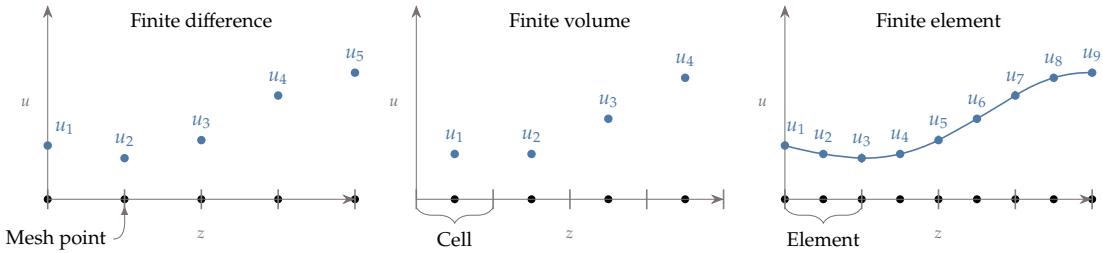
$$m = \sum_{i=1}^{15} \rho a_i l_i, \quad (3.6)$$

where  $\rho$  is the material density,  $a_i$  is the cross-sectional area of each member, and  $l_i$  is the member length.

---

### 3.4 Discretization of Differential Equations

Many physical systems are modeled by differential equations defined over a domain. The domain can be spatial (one or more dimensions), temporal, or both. When time is considered, then we have a dynamic model. When a differential equation is defined in a domain with one degree of freedom (1D in space or time), then we have an ordinary



differential equation (ODE), while any domain defined by more than one variable results in a partial differential equation (PDE).

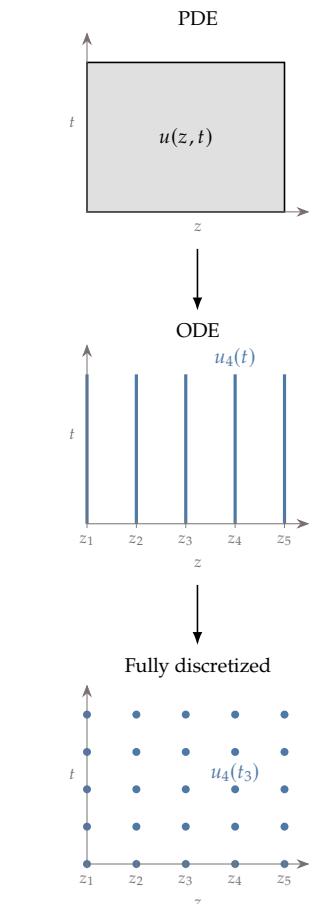
Differential equations need to be discretized over the domain to be solved numerically. There are three main methods for the discretization of differential equations: the finite-difference method, the finite-volume method, and the finite-element method. The finite-difference method approximates the derivatives in the differential equations by the value of the relevant quantities at a discrete number of points in a mesh (see Fig. 3.3). The finite-volume method is based on the integral form of the PDEs. It divides the domain into control volumes called cells (which also form a mesh), and the integral is evaluated for each cell. The values of the relevant quantities can be defined either at the centroids of the cells or at the cell vertices. The finite-element model divides the domain into elements (which are similar to cells) over which the quantities are interpolated using pre-defined shape functions. The values are computed at specific points in the element that are not necessarily at the element boundaries. Governing equations can also include integrals, which can be discretized with quadrature rules.

With any of these discretization methods, the final result is a set of algebraic equations that we can write as  $r(u) = 0$  and solve for the state variables  $u$ . This is a potentially large set of equations depending on the domain and discretization (it is common to have millions of equations in three-dimensional computational fluid dynamic problems). The number of state variables of the discretized model is equal to the number of equations for a complete and well-defined model. In the most general case, the set of equations could be implicit and nonlinear.

When a problem involves both space and time, the prevailing approach is to decouple the discretization in space from the discretization in time—called the method of lines (see Fig. 3.4). The discretization in space is performed first, yielding an ODE in time. The time derivative can then be approximated as a finite difference, leading to a time-integration scheme.

The discretization process usually yields implicit algebraic equations that require a solver to obtain the solution. However, discretization

**Figure 3.3:** Discretization methods in one spatial dimension.



**Figure 3.4:** PDEs in space and time are often discretized in space first to yield an ODE in time.

in some cases yield explicit equations, in which case a solver is not required.

### 3.5 Numerical Errors

Numerical errors (or computation errors) can be categorized into three main types: round-off errors, truncation errors, and errors due to coding. Numerical errors are involved with each of the modeling steps between the mathematical model and the states (see Fig. 3.1). The error involved in the discretization step is a type of truncation error. The errors introduced in the coding step are not usually discussed as a numerical error, but we include them here because they are a likely source of error in practice. The errors in the computation step involve both round-off and truncation errors. Each of these error sources is discussed in the following subsections.

An *absolute error* is the magnitude of the difference between the exact value ( $x^*$ ) and the computed value ( $x$ ), which we can write as  $|x - x^*|$ . The *relative error* is a more intrinsic error measure and is defined as

$$\epsilon = \frac{|x - x^*|}{|x^*|}. \quad (3.7)$$

This is more useful error measure in most cases. When the exact value  $x^*$  is close to zero, however, this definition breaks down. To address this, we avoid the division by zero by using

$$\epsilon = \frac{|x - x^*|}{1 + |x^*|}. \quad (3.8)$$

This error metric combines the properties of absolute and relative errors. When  $|x^*| \gg 1$ , this metric is similar to the relative error, but when  $|x^*| \ll 1$ , it becomes similar to the absolute error.

#### 3.5.1 Roundoff Errors

Roundoff errors stem from the fact that a computer cannot represent all real numbers with exact precision. Errors in the representation of each number lead to errors in each arithmetic operation, which in turn might accumulate over the course of a program.

There are an infinite number of real numbers, but not all numbers can be represented in a computer. When a number cannot be represented exactly, it is rounded. In addition, a number might be too small or too large to be represented.

Computers use bits to represent numbers, where each bit is either 0 or 1. Most computers use the IEEE standard for representing numbers and performing finite-precision arithmetic. The most common representation uses 32 bits for integers and 64 bits for real numbers.

Basic operations that only involve integers and whose result is an integer do not incur numerical errors. However, there is a limit on the range of integers that can be represented. When using 32 bit integers, 1 bit is used for the sign, and the remaining 31 bits can be used for the digits, which results in a range from  $-2^{31} = -2,147,483,648$  to  $2^{31}-1 = 2,147,483,647$ . Any operation outside this range would result in *integer overflow*.\*

Real numbers are represented using scientific notation in base 2

$$x = \text{significand} \times 2^{\text{exponent}} \quad (3.9)$$

The 64-bit representation is known as the double-precision floating-point format, where some digits store the significand and others store the exponent. The greatest positive and negative real numbers that can be represented using the IEEE double-precision representation are approximately  $10^{308}$  and  $-10^{308}$ , respectively. Operations that result in numbers outside this range result in *overflow*, which is a fatal error in most computers and interrupts the program execution.

There is also a limit on how close a number can come to zero, which is approximately  $10^{-324}$  when using double precision. Numbers smaller than this result in *underflow*. The computer sets such numbers to zero by default and the program usually proceeds with no harmful consequences.

One important number to consider in roundoff error analysis is the *machine precision*,  $\epsilon_{\text{mach}}$ , which represents the precision of the computer calculations. This is the smallest positive number  $\epsilon$  such that

$$1 + \epsilon > 1 \quad (3.10)$$

when calculated using a computer. Typically double precision divides up the 64 bits representation with 1 bit for the sign, 11 bits for the exponent, and 52 bits for the significand. Thus, when using double precision,  $\epsilon_{\text{mach}} = 2^{-52} \approx 2.2 \times 10^{-16}$ . Thus, a double-precision number has about 16 digits of precision and a relative representation error of up to  $\epsilon_{\text{mach}}$  may occur.

---

**Example 3.4:** Machine precision

Suppose that three decimal digits are available to represent a number (and that we use base 10 for simplicity). Then,  $\epsilon_{\text{mach}} = 0.005$ , because any

\*Some programming languages, such as Python, have arbitrary precision integers and are not subject to this issue, albeit with some performance tradeoffs.

number smaller than this results in  $1 + \epsilon = 1$  when rounded to three digits. For example,  $1.00 + 0.00499 = 1.00499$ , which rounds to 1.00. On the other hand,  $1.00 + 0.005 = 1.005$ , which rounds to 1.01 and satisfies Eq. 3.10.

### Example 3.5: Relative representation error

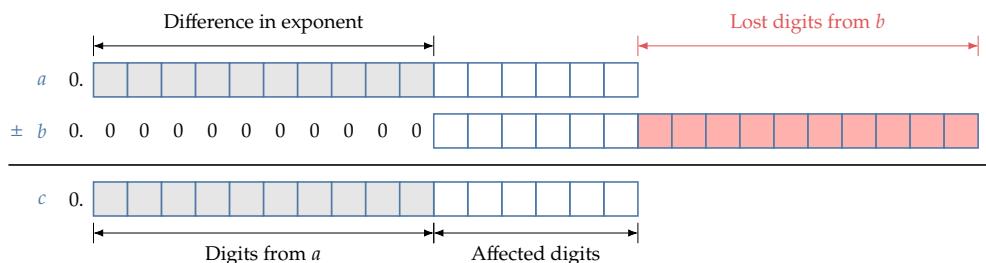
If we try to store 24.11 using three digits, we get 24.1. The relative error is

$$\frac{24.11 - 24.1}{24.11} \approx 0.0004, \quad (3.11)$$

which is lower than the maximum possible representation error of  $\epsilon_{\text{mach}} = 0.005$  established in Ex. 3.4

When performing an operation with numbers that contain errors, the result is subject to a *propagated error*. For multiplication and division, the relative propagated error is approximately the sum of the relative errors of the respective two operands.

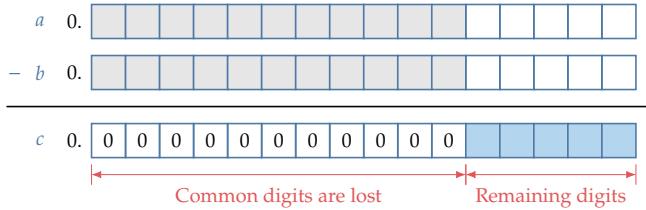
For addition and subtraction, an error can occur even when the two operands are represented exactly. Before addition and subtraction, the computer must convert the two numbers to the same exponent. When adding numbers with different exponents, several digits from the small number will vanish (see Fig. 3.5). If the difference in the two exponents is greater than the magnitude of the exponent of  $\epsilon_{\text{mach}}$ , the small number will vanish completely—a consequence of Eq. 3.10. The relative error incurred in addition is still around  $\epsilon_{\text{mach}}$ .



Subtraction, on the other hand, can incur much greater relative errors when subtracting two numbers that have the same exponent and are close to each other. In this case, the digits that match between the two numbers cancel each other and reduce the number of significant digits. When the relative difference between two numbers is less than machine precision, all digits match and the subtraction result is zero (see

**Figure 3.5:** Adding or subtracting numbers of differing exponents results in a loss in the number of digits corresponding to the difference in the exponents.

Fig. 3.6). This is called *subtractive cancellation* and is a prevailing issue when approximating derivatives via finite differences (see Section 6.4).

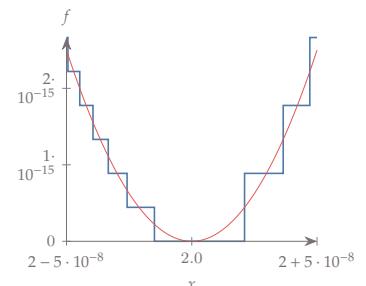


**Figure 3.6:** Subtracting two numbers that are close to each other results in a loss of the digits that match.

Sometimes, small roundoff errors can propagate and result in much larger errors. This can happen when a problem is ill-conditioned or when the algorithm used to solve the problem is unstable. In both cases, small changes in the inputs cause large changes in the output. Ill-conditioning is not a consequence of the finite-precision computations, but is a characteristic of the model itself. Stability is a property of the algorithm used to solve the problem. When a problem is ill conditioned, it is challenging to solve it in a stable way. When a problem is well conditioned, there is a stable way to solve it, but there may still be algorithms that are unstable.

#### Example 3.6: Effect of roundoff error on function representation

Let us examine the function  $f(x) = x^2 - 4x + 4$  near the minimum which is at  $x = 2$ . If we use double precision and plot many points in a small interval, we can see that the function exhibits the step pattern shown in Fig. 3.7. The numerical minimum of this function is anywhere in the interval around  $x = 2$  where the numerical value is zero. This interval is much larger than the machine precision ( $\epsilon_{\text{mach}} = 2.2 \times 10^{-16}$ ) An additional error is incurred in the function computation around  $x = 2$  due to subtractive cancellation.



### 3.5.2 Truncation Errors

In the most general sense, truncation errors arise from performing a finite number of operations where an infinite number of operations would be required to get an exact result.<sup>†</sup> Truncation errors would arise even if we could do the arithmetic with infinite precision.

When discretizing a mathematical model with partial derivatives, these are approximated by truncated Taylor series expansions that ignored higher-order terms. When the model includes integrals, they are approximated as finite sums. In either case, a mesh of points where the relevant states and functions are evaluated is required.

**Figure 3.7:** With double precision, the minimum of this quadratic function is in an interval much larger than machine zero.

<sup>†</sup>Roundoff error, discussed in the previous section, is sometimes also referred to as truncation error because digits are *truncated*, but we avoid this confusing naming and only use truncation error to refer to a truncation in the number of operations.

Discretization errors generally decrease as the spacing between the points decreases.

**Tip 3.7:** Perform a mesh refinement study

When using a model that depend on a mesh, it is a good idea to perform a mesh refinement study. This involves solving the model for increasingly finer meshes to check if the metrics of interest converge in a stable way and to verify that the converge rate is as expected for the numerical discretization scheme that is used. Such a study is also useful to find out which mesh provides the best compromise between computational time and accuracy, which is especially important in numerical optimization because of the number of times that the model is solved.

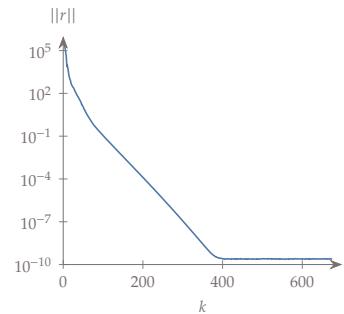
### 3.5.3 Iterative Solver Tolerance Error

Many methods for solving numerical models involve an iterative procedure that starts with a guess for the states  $u$  and then improve that guess at each iteration until reaching a specified convergence tolerance. The convergence is usually measured by a norm of the residuals,  $\|r(u)\|$ , which we want to drive to zero. Iterative linear solvers and Newton-type solvers are examples of iterative methods (see Section 3.7).

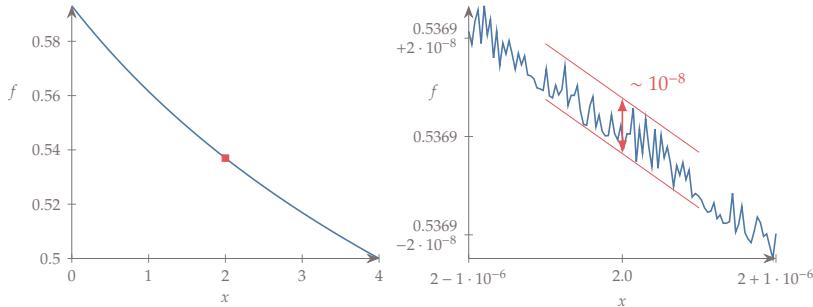
A typical convergence history for an iterative solver is shown in Fig. 3.8. The norm of the residuals decreases gradually until a limit is reached (near  $10^{-10}$  in this case). This limit represents the lowest error that can be achieved with the iterative solver and is determined by other sources of error such as roundoff and truncation errors. If we terminate before reaching the limit (either by setting a convergence tolerance to a value higher than  $10^{-10}$ , or by setting an iteration limit to lower than 400 iterations), we incur an additional error. However, it might be desirable to trade-off a less precise solution for a lower computational effort.

**Tip 3.8:** Find the level of the numerical noise in your model.

It is important to know the level of error in your model because this limits the type of optimizer you can use and how well you can optimize. In Ex. 3.6, we saw that if we plot a function at a small enough scale, we can see discrete steps in the function due to roundoff errors. When accumulating all sources of error in a more elaborate model (roundoff, truncation, and iterative), we no longer have a neat step pattern. Instead, we get *numerical noise*, as shown in Fig. 3.9. The level of noise can be estimated by the amplitude of the oscillations and gives us the order of magnitude of the total numerical error.



**Figure 3.8:** Norm of residuals versus the number of iterations for an iterative solver.



**Figure 3.9:** To find the level of numerical noise of a function of interest with respect to an input parameter (left), we magnify both axes by several orders of magnitude and evaluate the function at points that are closely spaced (right).

### 3.5.4 Programming Errors

Most of the literature on numerical methods is too optimistic and does not explicitly discuss programming errors, which are commonly known as bugs. Most programmers, especially beginners, underestimate the likelihood that their code has bugs.

It is helpful to adopt sound programming practices, such as writing clear code that is modular. Clear code has consistent formatting, meaningful naming of variable functions, and helpful comments. Modular code re-uses and generalizes functions as much as possible and avoids copying and pasting sections of code.<sup>51</sup>

There are different types of bugs that are relevant to numerical models: generic programming errors, incorrect memory handling, and algorithmic or logical errors.

Programming errors are the most frequent and include typos, type errors, copy-and-paste errors, faulty initialization, missing switch cases, and default values. In theory, these errors can be avoided through careful programming and code inspection, but in practice, you must always test your code. The testing involves comparing your result with for a case where you know the solution—the reference result. You should start with the simplest representative problem and then build up from that. An interactive debugger is a helpful tool that enables you to observe what the code is doing at runtime and check intermediate variable values.

**Tip 3.9:** Assume there are bugs in your code.

The overall attitude towards programming should be that all code has bugs until it is verified through testing.

Memory handling issues are much less frequent than programming errors, but they are usually more difficult to track. These issues include memory leaks (a failure to free unused memory), incorrect use of

<sup>51.</sup> Wilson et al., *Best Practices for Scientific Computing*, 2014

memory management, buffer overruns (such as array bound violations), and reading uninitialized memory. Memory issues are difficult to track because they can result in strange behavior in parts of the code that are far from the source of the error. In addition, they might manifest themselves in specific conditions that are hard to reproduce consistently. Memory debuggers are essential tools for addressing memory issues. They perform a detailed bookkeeping of all allocation, deallocation, and memory access to detect and locate any irregularities.<sup>‡</sup>

Whereas programming errors are due to a mismatch between the programmer's intent and what is coded, the root cause of algorithmic or logical errors is in the programmer's intent itself. Again, testing is the key to finding these errors, but you must be sure that the reference result is correct.

Running the analysis within an optimization loop has the potential to reveal bugs that do not manifest themselves in a single analysis. Therefore, after (and only after!) you have tested the analysis code in isolation, should you run an optimization test case.

As previously mentioned, there is a higher incentive to reduce the computational cost of an analysis when it runs in an optimization loop because it will run many times. When you first write your code, you should prioritize clarity and correctness as opposed to speed. Once the code is verified through testing, you should identify any bottlenecks using a performance profiling tool. Memory performance issues can also arise from running the analysis in an optimization loop as opposed to running a single case. In addition to running a memory debugger, you can also run a memory profiling tool to identify opportunities to reduce memory usage.

### 3.6 Rate of Convergence

Iterative solvers compute a *sequence* of approximate solutions that hopefully converge to the exact solution. When characterizing convergence, we need to first establish if the algorithm converges and if so, how fast does it converge. The first characteristic relates to the stability of the algorithm. Here, we focus on the second characteristic, which is quantified through the *rate of convergence*.

The cost of iterative algorithms is often measured by counting the number of iterations required to achieve the solution. Iterative algorithms often require an infinite number of iterations to converge to the exact solution. In practice, we want to converge to an approximate solution that is close enough to the exact one. Determining the rate of

<sup>‡</sup>See Grotker *et al.*<sup>52</sup> for more details on how to debug and profile code.

<sup>52</sup>. Grotker *et al.*, *The Developer's Guide to Debugging*. 2012

convergence arises from the need to quantify how fast the approximate solution is approaching the exact one.

In the following, we assume that we have a sequence of points,  $x^{(0)}, x^{(1)}, \dots, x^{(k)}, \dots$  that represent approximate solutions in the form of vectors in any dimension, and converge to a solution  $x^*$ . Then,

$$\lim_{k \rightarrow \infty} \|x^{(k)} - x^*\| = 0, \quad (3.12)$$

which means that the norm of the error tends to zero as the number of iterations tends to infinity. This sequence converges with *order r* when  $r$  is the largest number that satisfies

$$0 \geq \lim_{k \rightarrow \infty} \frac{\|x^{(k+1)} - x^*\|}{\|x^{(k)} - x^*\|^r} = \gamma < \infty, \quad (3.13)$$

where  $r$  is the *asymptotic convergence rate*, and  $\gamma$  is the *asymptotic convergence error constant*. “Asymptotic” here refers to the fact that we care mostly about the behavior in the limit. There is no guarantee that the initial and intermediate iterations satisfy this condition.

To avoid dealing with limits, let us assume that condition (3.13) applies to all iterations. We can relate the error from one iteration to the next by

$$\|x^{(k+1)} - x^*\| = \gamma \|x^{(k)} - x^*\|^r. \quad (3.14)$$

When  $r = 1$ , we have *linear* convergence, and when  $r = 2$ , we have *quadratic* convergence. Quadratic convergence is a highly valued property for an algorithm and in practice higher rates of convergence are usually not considered.

When we have linear convergence, then

$$\|x^{(k+1)} - x^*\| = \gamma \|x^{(k)} - x^*\|. \quad (3.15)$$

In this case the convergence is highly dependent on the value of the asymptotic constant  $\gamma$ . If  $\gamma > 1$ , then the sequence diverges—a situation to be avoided. If  $0 < \gamma < 1$ , then the norm of the error decreases by a constant factor for every iteration. If  $\gamma = 0.1$ , for example, and we start with an initial error norm of 0.1, we get the sequence,

$$10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, \dots \quad (3.16)$$

Thus, after six iterations, we get six-digit accuracy. Now suppose that  $\gamma = 0.9$ . Then we would have

$$\begin{aligned} 10^{-1}, 9.0 \times 10^{-2}, 8.1 \times 10^{-2}, 7.3 \times 10^{-2}, 6.6 \times 10^{-2}, \\ 5.9 \times 10^{-2}, 5.3 \times 10^{-2}, \dots, \end{aligned} \quad (3.17)$$

which corresponds to only one-digit accuracy after six iterations. It would take 131 iterations to achieve the same six-digit accuracy.

When we have quadratic convergence, then

$$\|x^{(k+1)} - x^*\| = \gamma \|x^{(k)} - x^*\|^2. \quad (3.18)$$

If  $\gamma = 1$ , then the error norm sequence with a starting error norm of 0.1 would be

$$10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, \dots \quad (3.19)$$

Thus we achieve more than six digits of accuracy in just three iterations! In this case, the number of correct digits doubles at every iteration. For  $\gamma > 1$ , the convergence might not be as good, but the series is still convergent.

If  $r \geq 1$  and  $\gamma \rightarrow 0$ , we have *superlinear* convergence, which includes quadratic and higher rates of convergence. There is a special case of superlinear convergence that is relevant for optimization algorithms, which is when  $r = 1$ . This case is desirable because in practice it behaves similarly to quadratic convergence and can be achieved by gradient-based algorithms that use first derivatives (as opposed to second derivatives). In this case, we can write

$$\|x^{(k+1)} - x^*\| = \gamma^{(k)} \|x^{(k)} - x^*\|, \quad (3.20)$$

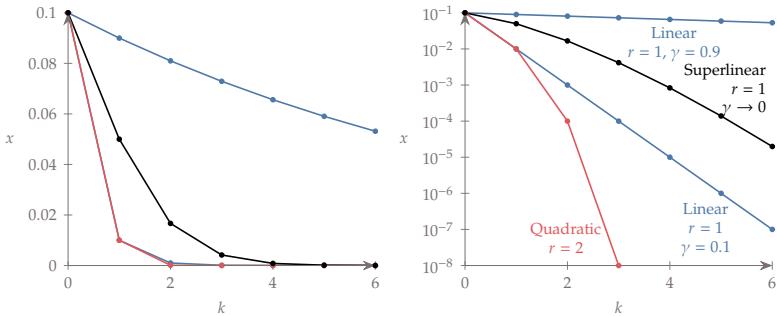
where  $\lim_{k \rightarrow \infty} \gamma^{(k)} = 0$ . Now we need to consider a sequence of values for  $\gamma$  that tends to zero. For example, if  $\gamma = 1/(k + 1)$ , starting with an error norm of 0.1, we get,

$$\begin{aligned} 10^{-1}, 5 \times 10^{-1}, 1.7 \times 10^{-1}, 4.2 \times 10^{-2}, 8.3 \times 10^{-4}, \\ 1.4 \times 10^{-4}, 2.0 \times 10^{-5}, \dots \end{aligned} \quad (3.21)$$

Thus, we have achieved four-digit accuracy after six iterations. This special case of superlinear convergence is not quite as good as quadratic convergence, but it is better than either of the linear convergence examples above.

We plot the sequences above in Fig. 3.10. Since the points are just scalars and the exact solution is zero, the norm or the error is just  $x^{(k)}$ . The first plot uses a linear scale so we cannot see any differences beyond two digits. To examine the differences more carefully, we need to use a logarithmic axis for the sequence values, as shown on the right plot. In this scale, each decrease in order of magnitude represents one more digit of accuracy. We can see how the linear convergence shows up as a straight line in this plot, but the slope of the line varies widely, depending on the value of the asymptotic constant. Quadratic

convergence exhibits an increasing slope, reflecting the doubling of digits for each iteration. the superlinear sequence exhibits poorer convergence than the best linear one, but we can see that the slope of the superlinear curve is increasing, which means that for a high enough  $k$ , it will converge at a higher rate than the linear one.



**Figure 3.10:** Sample sequences for linear, superlinear, and quadratic cases plotted in a linear scale (left) and logarithmic scale (right).

---

**Tip 3.10:** Use a logarithmic scale when plotting convergence

When using a linear scale plot, you can only see differences in two significant digits. To reveal changes beyond three digits, you should use a logarithmic scale. This need occurs frequently in plotting the convergence behavior of optimization algorithms.

---

When solving numerical models we can monitor the norm of the residual. Because we know that the residuals should be zero for an exact solution, the norm of the error is simply the norm of the residual,  $\|r(u^k)\|$ .

If we monitor another quantity, we do not usually know the exact solution. In these cases, we can use the ratio of the step lengths of each iteration:

$$\frac{\|x^{(k+1)} - x^*\|}{\|x^{(k)} - x^*\|} \approx \frac{\|x^{(k+1)} - x^{(k)}\|}{\|x^{(k)} - x^{(k-1)}\|}, \quad (3.22)$$

The rate of convergence can then be estimated numerically with the values of the last available four iterates using

$$r \approx \frac{\log \frac{\|x^{(k+1)} - x^{(k)}\|}{\|x^{(k)} - x^{(k-1)}\|}}{\log \frac{\|x^{(k)} - x^{(k-1)}\|}{\|x^{(k-1)} - x^{(k-2)}\|}}. \quad (3.23)$$

Finally, we can also monitor any quantity by taking the step length

and normalizing it in the same way as Eq. 3.8,

$$\frac{\|x^{(k+1)} - x^{(k)}\|}{1 + \|x^{(k)}\|}. \quad (3.24)$$

### 3.7 Overview of Solvers

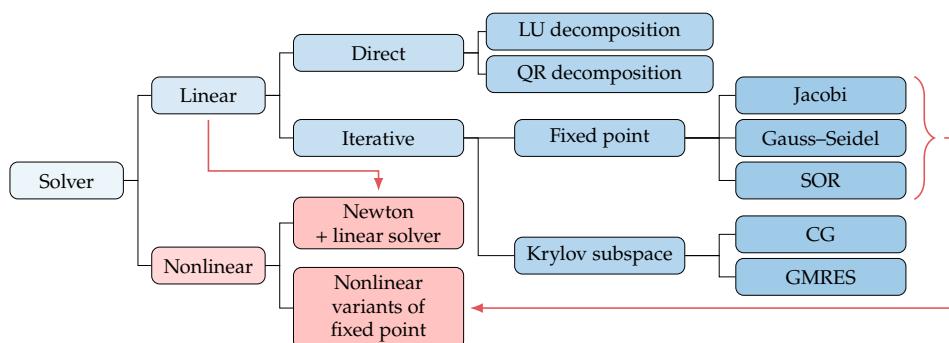
There are a number of methods available for solving the discretized governing equations (3.1). We want to solve the governing equations for a fixed set of design of design variables, so  $x$  will not appear in the solution algorithms. Our objective is to find the state variables  $u$  such that  $r(u) = 0$ .

This is not a book about solvers, but it is important to understand the characteristics of these solvers because they affect the cost and precision of the function evaluations in the overall optimization process. Thus, we provide an overview and some of the most relevant details in this section. In addition, the solution of coupled systems builds on these solvers, as we will see in Section 13.3. Finally, some of the optimization algorithms detailed in later chapters use these solvers.  $\S$

There are two main types of solvers, depending on whether the equations to be solved are linear or nonlinear (Fig. 3.11). Linear solution methods solve systems of the form  $r(u) = Au - b = 0$ , where the matrix  $A$  and vector  $b$  are not dependent on  $u$ . Nonlinear methods can handle any algebraic system of equations that can be written as  $r(u) = 0$ .

$\S$ Ascher *et al.*<sup>53</sup> provides a more detailed introduction to the numerical methods mentioned in this chapter.

53. Ascher *et al.*, *A first course in numerical methods*. 2011



Linear systems can be solved directly or iteratively. Direct methods are based on the concept of Gaussian elimination, which can be expressed in matrix form as a decomposition into lower and upper triangular matrices that are easier to solve (*LU* decomposition). Cholesky decomposition is a variant of *LU* decomposition that applies only to symmetric positive-definite matrices.

Figure 3.11: Overview of solution methods for linear and nonlinear systems.

While direct solvers obtain the solution  $u$  at the end of a process, iterative solvers start with a guess for  $u$  and successively improve it with each iteration through explicit expressions that are easy to compute, as illustrated in Fig. 3.12. Iterative methods can be fixed-point iterations, such as Jacobi, Gauss–Seidel, and successive over-relaxation (SOR), or Krylov subspace methods, such as the conjugate gradient (CG) and generalized minimum residual (GMRES) methods.<sup>¶</sup> Direct solvers are well established and are included in the standard libraries for most programming languages. Iterative solvers are less widespread in standard libraries, but they are becoming more commonplace.

**Tip 3.11:** Do not compute the inverse of  $A$ .

Because some numerical libraries have functions to compute  $A^{-1}$ , you might be tempted to do this and then multiply by a vector to compute  $u = A^{-1}b$ . This is a bad idea because finding the inverse is computationally expensive. Instead, use  $LU$  decomposition or another method from Fig. 3.11.

Direct methods are the right choice for many problems because they are generally robust. However, for large systems where  $A$  is sparse, the cost of direct methods can become prohibitive, while iterative methods remain viable. Iterative methods have other advantages, such as being able to trade between computational cost and precision, and to restart from a good guess (see Appendix B for details).

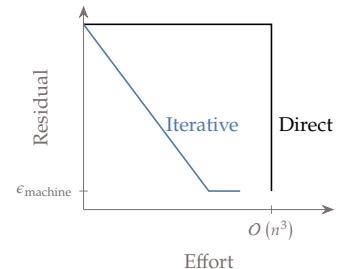
When it comes to nonlinear solvers, the most efficient methods are based on Newton’s method, which we explain later in this chapter (Section 3.8). Newton’s method solves sequence of problems that are linearizations of the nonlinear problem about the current iterate. The linear problem at each Newton iteration can be solved using any linear solver, as indicated by the incoming arrow in Fig. 3.11. Although efficient, Newton’s method is not robust in that it does not always converge. Therefore, it requires modifications so that it can converge reliably.

Finally, it is possible to adapt linear fixed point iterations methods to solve nonlinear equations as well. However, unlike the linear case, it might not be possible to derive explicit expressions for the iterations in the nonlinear case. For this reason, fixed point iteration methods are not usually the best choice for solving a system of nonlinear equations. However, as we will see in Section 13.3.4, these methods are useful for solving of systems of coupled nonlinear equations.

For time dependent problems, we require a way to solve for the time history of the states,  $u(t)$ . As mentioned in Section 3.3, the most

<sup>¶</sup>See Saad<sup>54</sup> for more details on iterative methods in the context of large-scale numerical models.

<sup>54</sup>. Saad, *Iterative Methods for Sparse Linear Systems*. 2003



**Figure 3.12:** While direct methods only yield the solution at the end of the process, iterative methods produce approximate intermediate results.

popular approach is to decouple the temporal discretization from the spatial one. By discretizing a PDE in space first, this method formulates an ODE in time of the form,

$$\frac{du}{dt} = -r(u), \quad (3.25)$$

which is called the semi-discrete form. A time-integration scheme is then used to solve for the time history. A time-integration scheme can be either explicit or implicit, depending on whether it involves evaluating explicit expressions, or requires solving implicit equations. If a system under a certain condition has a steady state, these techniques can be used to solve for the steady state ( $du/dt = 0$ ).

### 3.8 Newton-based Solvers

As mentioned in Section 3.7, Newton's method is the basis for many nonlinear equation solvers. Newton's method is also at the core of the most efficient gradient-based optimization algorithms, so we explain it here in more detail. We start with the single-variable case for simplicity, and then generalize it to the  $n$ -dimensional case.

We want to find  $u^*$  such that  $r(u^*) = 0$ , where for now,  $r$  and  $u$  are scalars. Newton's method estimates a solution at each iteration  $u^{(k)}$  by approximating  $r(u^{(k)})$  to be a linear function. The linearization is done by taking a Taylor's series of  $r$  about  $u^{(k)}$  and truncating it to obtain

$$r(u^{(k)} + \Delta u) \approx r(u^{(k)}) + \Delta u r'(u^{(k)}), \quad (3.26)$$

where  $r' \triangleq dr/du$  and  $r^{(k)} \triangleq r(u^{(k)})$ . Now we can use this to find the step  $\Delta u$  that makes approximate residual zero,

$$r^{(k)} + \Delta u r'^{(k)} = 0 \quad \Rightarrow \quad \Delta u = -\frac{r^{(k)}}{r'^{(k)}}. \quad (3.27)$$

where we need to assume that  $r'^{(k)} \neq 0$ .

Thus, the update for each step in Newton's algorithm is

$$u^{(k+1)} = u^{(k)} - \frac{r^{(k)}}{r'^{(k)}}. \quad (3.28)$$

If  $r'^{(k)} = 0$ , the algorithm will not converge because it yields a step to infinity. Small enough values of  $r'^{(k)}$  also cause an issue with large steps, but the algorithm might still converge.

---

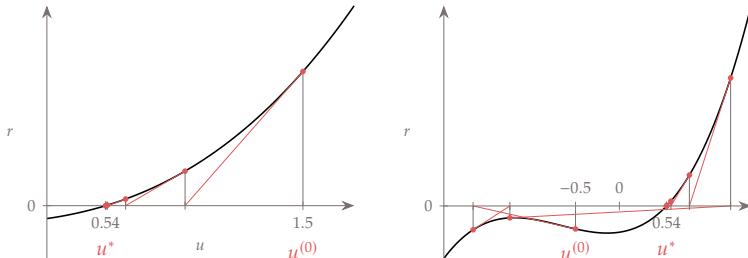
**Example 3.12:** Newton's method for a single variable

Suppose we want to solve the equation  $r(u) = 2u^3 + 4u^2 + u - 2 = 0$ . Since  $r'(u) = 6u^2 + 8u + 1$ , the Newton iteration is,

$$u^{(k+1)} = u^{(k)} - \frac{2u^{(k)^3} + 4u^{(k)^2} + u^{(k)} - 2}{6u^{(k)^2} + 8u^{(k)} + 1}. \quad (3.29)$$

When we start with the guess  $u^{(0)} = 1.5$  (left plot), the iterations are well behaved and the method converges quadratically. We can see a geometric interpretation of Newton's method: For each iteration, it takes the tangent to the curve and finds the intersection with  $r = 0$ .

When we start with  $u^{(0)} = 1.5$ , the first step goes in the wrong direction, but recovers in the second iteration. The third iteration is close to the point with zero derivative and takes a large step. In this case, the iterations recover and then converge normally. However, we can easily envision a case where an iteration is much closer to the point with zero derivative, causing an arbitrarily long step.



**Figure 3.13:** Newton iterations starting from different starting points.

---

Newton's method converges quadratically when close enough to the solution with a convergence constant of

$$\gamma = \left| \frac{r''(u^*)}{2r'(u^*)} \right|. \quad (3.30)$$

This means that if the derivative is close to zero or the curvature tends to a large number at the solution, Newton's method will not converge as well or not at all.

Now we consider the general case where we have  $n$  nonlinear equations of  $n$  unknowns, expressed as  $r(u) = 0$ . Similarly to the single-variable case, we derive the Newton step from a truncated Taylor series. However, the Taylor series needs to be multidimensional in both the independent variable and the function. Consider first the multidimensionality of the independent variable,  $u$ , for a component of the residuals,  $r_i(u)$ . The first two terms of the Taylor series about  $u^{(k)}$  for a step  $\Delta u$  (which is now a vector with arbitrary direction and

magnitude) are

$$r_i(u^{(k)} + \Delta u) \approx r_i(u^{(k)}) + \sum_{j=1}^n \Delta u_j \frac{\partial r_i}{\partial u_j} \Big|_{u=u^{(k)}}. \quad (3.31)$$

Since we have  $n$  residuals,  $i = 1, \dots, n$ , and we can write the second term in matrix form as  $J\Delta u$ , where  $J$  is a square matrix whose elements are

$$J_{ij} = \frac{\partial r_i}{\partial u_j}. \quad (3.32)$$

This is called the *Jacobian matrix*.

Similarly to the single-variable case, we want to find the step that makes the two terms zero, which yields the linear system,

$$J^{(k)} \Delta u^{(k)} = -r^{(k)}, \quad (3.33)$$

After solving this linear system, we can update the solution to

$$u^{(k+1)} = u^{(k)} + \Delta u^{(k)}. \quad (3.34)$$

Thus, Newton's method involves solving a sequence of linear systems given by Eq. 3.33. The linear system and can be solved using any of the methods for solving linear systems mentioned in Section 3.7. One popular option for solving for the Newton step the Krylov method, which results in the Newton-Krylov method for solving nonlinear systems. Because the Krylov method only requires matrix-vector products of the form  $[\partial r / \partial u]v$ , we can avoid computing and storing the Jacobian by computing this product directly (using finite differences or other methods from Chapter 6).

The multivariable version of Newton's method is subject to the same issues we uncovered for the single-variable case: it only converges if the starting point is within a certain region and it can be subject to ill conditioning. Newton's method can be modified to increase the likelihood of convergence from any starting point, as we will see in Chapter 4. The ill-conditioning issue has to do with the linear system (3.33) and can be quantified by the condition number of the Jacobian matrix. Ill-conditioning can be addressed by scaling and preconditioning.

---

**Example 3.13:** Newton's method applied to two nonlinear equations.

Suppose we have the nonlinear system of two equations

$$u_2 = \frac{1}{u_1}, \quad u_2 = \sqrt{u_1}. \quad (3.35)$$

This corresponds to the two lines shown in Fig. 3.14, where the solution is at their intersection,  $u = (1, 1)$ . (In this example, the two equations are explicit and we could solve them by substitution, but they could have been implicit.)

To solve this using Newton's method, we need to write these as residuals,

$$\begin{aligned} r_1 &= u_2 - \frac{1}{u_1} = 0 \\ r_2 &= u_2 - \sqrt{u_1} = 0. \end{aligned} \quad (3.36)$$

The Jacobian can be derived analytically and the Newton step is given by the linear system

$$\begin{bmatrix} \frac{1}{u_1^2} & 1 \\ -\frac{1}{2\sqrt{u_1}} & 1 \end{bmatrix} \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \end{bmatrix} = - \begin{bmatrix} u_2 - \frac{1}{u_1} \\ u_2 - \sqrt{u_1} \end{bmatrix}. \quad (3.37)$$

Starting from  $u = (2, 3)$  yields the iterations shown below with the quadratic convergence shown in Fig. 3.15.

$u_1$	$u_2$	$\ u - u^*\ $	$\ r\ $
2.000000	3.000000	2.23	2.50
0.485281	0.878679	$5.28 \times 10^{-1}$	2.50
0.760064	0.893846	$2.62 \times 10^{-1}$	1.18
0.952668	0.982278	$5.05 \times 10^{-2}$	$4.21 \times 10^{-1}$
0.998289	0.999417	$1.80 \times 10^{-3}$	$6.74 \times 10^{-2}$
0.999998	0.999999	$2.31 \times 10^{-6}$	$2.29 \times 10^{-3}$
1.000000	1.000000	$3.81 \times 10^{-12}$	$2.93 \times 10^{-6}$
1.000000	1.000000	0.0	$4.83 \times 10^{-12}$
1.000000	1.000000	0.0	0.0

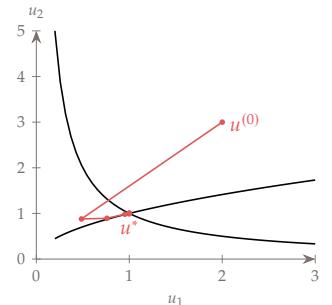


Figure 3.14: Newton iterations.

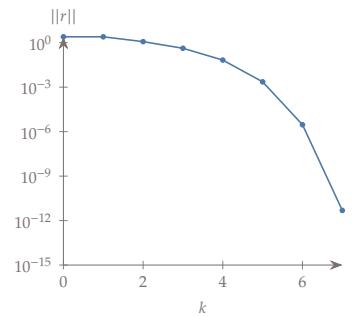


Figure 3.15: The norm of the residual exhibits quadratic convergence.

### 3.9 Models and the Optimization Problem

When performing design optimization, we ultimately need to compute the values of the objective and constraint functions in the optimization problem (1.4). There is typically an intermediate step that requires solving the governing equations for the given design  $x$  at one or more specific conditions. The governing equations define the state variables  $u$  as an implicit function of  $x$ , as illustrated in Fig. 3.16.

The objective and constraints are typically explicit functions of the state variables. In addition to depending implicitly on the design variables through the state variables, these functions can also include  $x$  explicitly, as indicated by the  $x$  in  $r(x, u) = 0$ . The dependency of these functions on the state and design variables is illustrated in Fig. 3.17, which is a more detailed version of Fig. 1.11. In design optimization

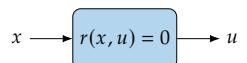


Figure 3.16: For a general model, the state variables  $u$  are implicit functions of the design variables  $x$  through the solution of the governing equations.

applications, the solution of the governing equations is usually the most computational intensive part of the optimization problem.

When we first introduced the general optimization problem (1.4), the governing equations were not included because they were assumed to be part of the computation of the objective and constraints for a given  $x$ . However, we can include them in the problem statement for completeness as follows

$$\begin{aligned}
 & \text{minimize} && f(x) \\
 & \text{by varying} && x_i \quad i = 1, \dots, n_x \\
 & \text{subject to} && g_j(x, u) \leq 0 \quad j = 1, \dots, n_g \\
 & && h_k(x, u) = 0 \quad k = 1, \dots, n_h \\
 & && \underline{x}_i \leq x_i \leq \bar{x}_i \quad i = 1, \dots, n_x \\
 & \text{while solving} && r_l(x, u) = 0 \quad l = 1, \dots, n_u \\
 & \text{by varying} && u_l \quad l = 1, \dots, n_u
 \end{aligned} \tag{3.38}$$

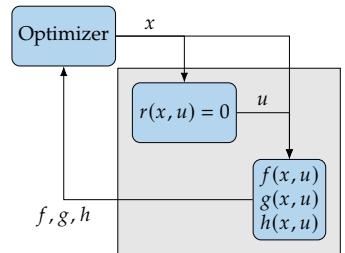
Here, “while solving” means that the governing equations are solved at each optimization iteration to find a valid  $u$  for each value of  $x$ .

**Example 3.14:** Structural sizing optimization.

Recalling the truss problem of Ex. 3.3, suppose we want to minimize the mass of the structure ( $m$ ) by varying the cross sectional areas of the trusses ( $x$ ), subject to stress constraints. We can write the problem statement as

$$\begin{aligned}
 & \text{minimize} && m(x) \\
 & \text{by varying} && x \geq x_{\min} \quad j = 1, \dots, 15 \\
 & \text{subject to} && |\sigma_j(x, u)| - \sigma_{\max} \leq 0 \quad j = 1, \dots, 15 \\
 & \text{while solving} && Ku - f = 0 \\
 & \text{by varying} && u_l \quad l = 1, \dots, 18
 \end{aligned} \tag{3.39}$$

The governing equations are a linear set of equations whose solution determines the displacements of the a given design ( $x$ ) for a given load condition ( $f$ ). We mentioned previously that the objective and constraint functions are usually explicit functions of the state variables, design variables, or both. As we saw in Ex. 3.3, the mass is indeed an explicit function of the cross sectional areas. In this case, it does not even depend on the state variables. The constraint function is also an explicit function, but in this case it is just a function of the state variables. This example illustrates a common situation where the solution of the state variables requires the solution of implicit equations (structural solver), while the constraints (stresses) and objective (weight) are explicit functions of the states and design variables.



**Figure 3.17:** The computation of the objective ( $f$ ) and constraint functions ( $h, g$ ) for a given set of design variables ( $x$ ) usually involves the solution of a numerical model ( $r = 0$ ) by varying the state variables ( $u$ ).

From a mathematical point of view, the model governing equations  $r(x, u) = 0$  can be considered equality constraints in an optimization problem. Some specialized optimization approaches add these equations to the optimization problem and let the optimization algorithm solve both the governing equations and optimization simultaneously. This is called a *full-space* approach, and is also known as simultaneous analysis and design (SAND) or one-shot optimization. The approach is stated as:

$$\begin{aligned}
 & \text{minimize} && f(x, u) \\
 & \text{by varying} && x_i \quad i = 1, \dots, n_x \\
 & && u_l \quad l = 1, \dots, n_u \\
 & \text{subject to} && g_j(x, u) \leq 0 \quad j = 1, \dots, n_g \\
 & && h_k(x, u) = 0 \quad k = 1, \dots, n_h \\
 & && \underline{x}_i \leq x_i \leq \bar{x}_i \quad i = 1, \dots, n_x \\
 & && r_l(x, u) = 0 \quad l = 1, \dots, n_u
 \end{aligned} \tag{3.40}$$

Unless otherwise stated, we assume that the optimization model governing equations are solved by a dedicated solver for each optimization iteration as stated in Eq. 3.38.

More generally, the optimization constraints and equations in a model are interchangeable. If a set of equations in a model can be satisfied by varying a corresponding set of state variables, these equations and variables can be moved to the optimization problem statement as equality constraints and design variables, respectively.

---

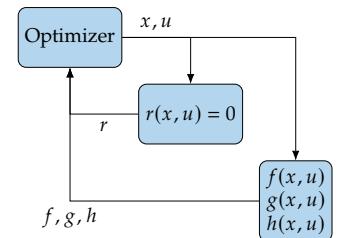
**Example 3.15:** Structural sizing optimization using a full-space approach.

---

If we wanted to solve this problem using a full-space approach, we would forgo the linear solver by adding  $u$  to the set of design variables and letting the optimizer enforce the governing equations. This would result in the following problem,

$$\begin{aligned}
 & \text{minimize} && m(x) \\
 & \text{by varying} && x_j \geq x_{\min} \quad j = 1, \dots, 15 \\
 & && u_l \quad l = 1, \dots, 18 \\
 & \text{subject to} && |\sigma_j(x, u)| - \sigma_{\max} \leq 0 \quad j = 1, \dots, 15 \\
 & && Ku - f = 0.
 \end{aligned} \tag{3.41}$$


---



**Figure 3.18:** In the full-space approach, the governing equations are solved by the optimizer by varying the state variables.

## 3.10 Summary

It is essential to understand the models that compute the objective and constraint functions because they directly impact the performance and

effectiveness of the optimization process.

The modeling process introduces several types of numerical errors associated with each step of the process (discretization, programming, computation). Knowing the level of numerical error is necessary to establish what precision can be achieved in the optimization. Understanding the types of errors involved helps us find ways to reduce those errors. Programming errors—“bugs”—are often underestimated; thorough testing is required to verify that the numerical model is coded correctly. A lack of understanding of a given model’s numerical errors is often the cause of the failure in optimization, especially when using gradient-based algorithms.

Modeling errors arise from discrepancies between the mathematical model and the real physical system. While they do not affect the optimization process’s performance and precision, modeling errors affect the accuracy and thus determine how valid the result is in the real world. Therefore, model validation and an understanding of modeling error are also critical.

In engineering design optimization problems, the models usually involve solving large sets of nonlinear implicit equations. The computational time required to solve these equations dominates the overall optimization time, and therefore, solver efficiency is crucial. Solver robustness is also crucial because optimization often asks for designs that are very different from what a human designer would ask for, which tests the limits of the model and the solver.

We presented an overview of the various types of solvers available for linear and nonlinear equations. Newton-type methods are highly desirable for solving nonlinear equations because they exhibit second-order convergence. Because Newton-type methods involve solving a linear system at each iteration, a linear solver is always required. These solvers are also at the core of several of the optimization algorithms in later chapters.

## Problems

### 3.1 Answer *true* or *false* and justify your answer.

- a) A model developed to perform well for analysis will generally do well in a numerical optimization process.
- b) Modeling errors have nothing to do with computations.
- c) Explicit and implicit equations can always be written in residual form.
- d) Subtractive cancellation is a type of roundoff error.

- e) Programming errors can be eliminated by carefully reading the code.
- f) Quadratic convergence is only better than linear convergence if the asymptotic convergence error constant is less or equal than one.
- g) Logarithmic scales are desirable when plotting convergence because they show errors of all magnitudes.
- h) Newton solvers always require a linear solver.
- i) Some linear iterative solvers can be used to solve nonlinear problems.
- j) Direct methods allow us to trade between computational cost and precision, while iterative methods do not.
- k) Newton's method requires the derivatives of all the state variables.
- l) In the full-space optimization approach, the state variables become design variables, and the governing equations become constraints.

3.2 Choose an engineering system that you are familiar with and describe each of the components illustrated in Fig. 3.1 for that system. List all the options for the mathematical and numerical models that you can think of and describe the assumptions for each model. What type of solver is usually used for each model (see Section 3.7) and what are their state variables? What are the state variables for each model?

3.3 Consider the following mathematical model:

$$\begin{aligned} u_1^2 + 2u_2 &= 1, \\ u_1 + \cos(u_1) - u_2 &= 0, \\ f(u_1, u_2) &= u_1 + u_2. \end{aligned}$$

Which equations are explicit and which ones are implicit? Write these equations in residual form.

3.4 Reproduce a plot similar to the one shown in Fig. 3.7 for

$$f(x) = \cos(x) + 1$$

in the neighborhood of  $x = \pi$ .

3.5 Using Newton's method, find the solution for

$$r(u) = u^3 - 6u^2 + 12u - 8 = 0.$$

Tabulate the residual for each iteration number. What is the lowest error you can achieve? Plot the residual versus the iteration number using a linear axis; how many digits can you discern in this plot? Make the same plot using a log axis for the residual and estimate the rate of convergence. *Exploration:* Try different starting points. Can you find a predictable trend?

- 3.6 Kepler's equation, which we mentioned in Section 2.2, defines the relationship between a planet's polar coordinates and the time elapsed from a given initial point and is

$$E - e \sin(E) = M,$$

where  $M$  is the mean anomaly (a parameterization of time),  $E$  is the eccentric anomaly (a parameterization of polar angle), and  $e$  is the eccentricity of the elliptical orbit. Use Newton's method to find  $E$  when  $e = 0.7$  and  $M = \pi/2$ . Devise a fixed-point iteration to solve the same problem. Compare the number of iterations and rate of convergence. *Exploration:* Plot  $E$  versus  $M$  in the interval  $[0, 2\pi]$  for  $e = [0, 0.1, 0.5, 0.9]$  and interpret your results physically.

- 3.7 Consider the equation from the previous exercise where we replace one of the coefficients with a parameter  $a$  as follows

$$r(u) = au^3 - 6u^2 + 12u - 8 = 0.$$

Find the level of the numerical noise in the solution  $u$  when solving this equation using Newton's method. Produce a plot similar to Fig. 3.9 by perturbing  $a$  in the neighborhood of  $a = 1$  using a solver convergence tolerance of  $|r| \leq 0.1$ . *Exploration:* Try smaller tolerances and see how much you can decrease the numerical noise.

- 3.8 Reproduce the solution of Ex. 3.13 and then try different initial guesses. Can you define a distinct region from where Newton's method converges?
- 3.9 Choose a problem that you are familiar with and find the magnitude of numerical noise in one or more outputs of interest with respect to one or more inputs of interest. What means do you have to decrease the numerical noise? What is the lowest possible level of noise you can achieve?

## Unconstrained Gradient-Based Optimization

# 4

In this chapter we focus our attention on unconstrained minimization problems with continuous design variables (see Fig. 1.12). Such optimization problems can be written as

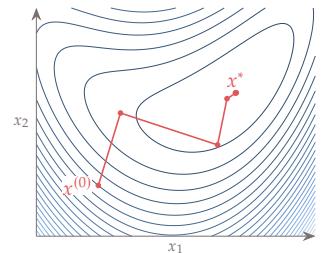
$$\begin{aligned} & \text{minimize } f(x) \\ & \text{by varying } x_i \quad i = 1, \dots, n_x, \end{aligned} \tag{4.1}$$

where  $x$  contains the design variables that the optimization algorithm can change. We will solve these problems using gradient information to determine a path from a starting guess (or baseline design) to the optimum, which consists of a series of discrete steps (see Fig. 4.1).

We assume the objective function to be nonlinear,  $C^2$  continuous, and deterministic. We make no assumption about multimodality and there is no guarantee that the algorithm finds the global optimum.

Referring to the attributes that classify an optimization problem (Fig. 1.19), the optimization algorithms discussed in this chapter range from first- to second-order, perform a local search, and evaluate the function directly. Both the iteration strategy (which is deterministic) and optimality criteria are based on mathematical principles as opposed to heuristics.

While most engineering design problems are constrained, the constrained optimization algorithms in Chapter 5 build on the techniques explained in the current chapter.



**Figure 4.1:** Gradient-based optimization starts with a guess,  $x^{(0)}$ , and takes a sequence of steps in  $n$ -dimensional space that converge to an optimum,  $x^*$ .

By the end of this chapter you should be able to:

1. Understand the significance of gradients, Hessians, and directional derivatives.
2. Understand the mathematical definition of optimality for an unconstrained problem.
3. Describe, implement, and use line-search-based methods.
4. Understand the pros and cons of various search direction methods.
5. Understand trust region approaches and how they contrast with line search methods.

## 4.1 Fundamentals

To determine the directions of the steps shown in Fig. 4.1, gradient-based methods need the gradient (first-order information). Some methods also use the curvature (second-order information). Gradients and curvature are required build a second-order Taylor series, which is a fundamental construct that is useful in establishing optimality and in developing gradient-based optimization algorithms.

### 4.1.1 Derivatives and Gradients

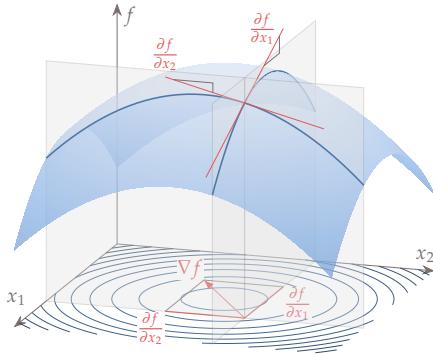
Recall that we are considering a scalar objective function  $f(x)$ , where  $x$  is the vector of design variables,  $x = [x_1, x_2, \dots, x_n]^T$ . The *gradient* of this function,  $\nabla f(x)$ , is a column vector of first-order partial derivatives of the function with respect to each design variable:

$$\nabla f(x) \equiv \left[ \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right]^T, \quad (4.2)$$

where each partial derivative is defined as the limit,

$$\frac{\partial f}{\partial x_i} \equiv \lim_{\epsilon \rightarrow 0} \frac{f(x_1, \dots, x_i + \epsilon, \dots, x_n) - f(x_1, \dots, x_n)}{\epsilon}. \quad (4.3)$$

Each component in the gradient vector quantifies the local rate of change of the function with respect to the corresponding design variable, as shown in Fig. 4.2 for the two-dimensional case. In other words, these components represent the slope of the function projected along each coordinate direction. The gradient is a vector pointing in the direction of greatest function increase from the current point.



**Figure 4.2:** Components of the gradient vector in the 2D case.

The gradient vectors are normal to contour surfaces of constant  $f$  in  $n$ -dimensional space. In the two-dimensional case, gradient vectors are perpendicular to the function contour lines, as shown in Fig. 4.2.

---

**Example 4.1:** Gradient of a polynomial function

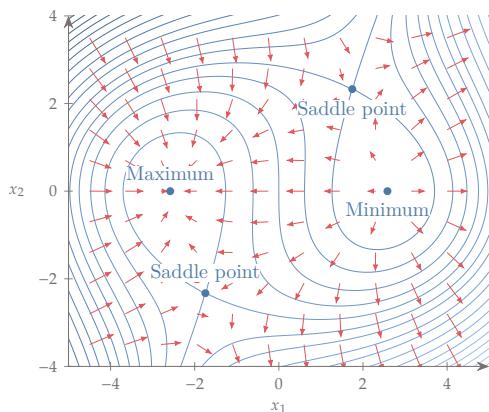
Consider the function of two variables,

$$f(x_1, x_2) = x_1^3 + 2x_1x_2^2 - x_2^3 - 20x_1 \quad (4.4)$$

The gradient can be obtained using analytic differentiation to obtain

$$\nabla f(x_1, x_2) = \begin{bmatrix} 3x_1^2 + 2x_2^2 - 20 \\ 4x_1x_2 - 3x_2^2 \end{bmatrix}. \quad (4.5)$$

This defines the vector field shown in Fig. 4.3, where each vector points in the direction of steepest local increase.



**Figure 4.3:** Gradient vector field shows how gradients point towards maxima and away from minima.

---

If a function is explicit function, we can use symbolic differentiation as we did in Ex. 4.1. However, recall from Chapter 3 that functions

can be the result of the solution of implicit equations. Closed form expressions for the derivatives of such equations is not possible, but there are established methods for computing them, which are the subject of Chapter 6.

Physically, each gradient component has units that correspond to the units of the function divided by the units of the corresponding variable. Since the variables might represent different physical quantities, each gradient component might have different units.

From the engineering design point of view, it might be useful to think about *relative* changes, where the derivative is given as the percentage change in the function for a 1% increase in the variable. This relative derivative can be computed by non-dimensionalizing both the function and the variable, yielding

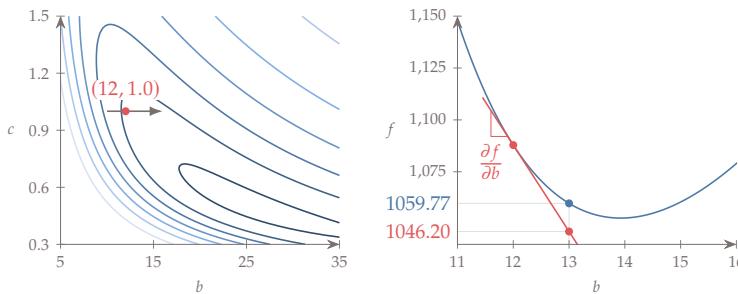
$$\frac{\partial f}{\partial x} \frac{x}{f}, \quad (4.6)$$

where  $f$  and  $x$  are the values of the function and variable at the point where the derivative is computed.

---

**Example 4.2:** Interpretation of derivatives for wing design problem.

Consider the wing design problem from Ex. 1.1, where the objective function  $f$  is the required power. For the derivative of power with respect to span ( $\partial f / \partial b$ ), the units are Watts per meter (W/m). For example, for a wing with  $c = 1$  m and  $b = 12$  m, we have  $f = 1087.85$  W and  $\partial f / \partial b = -41.65$  W/m. This means that an increase in span of 1 m a linear approximation predicts a decrease power of 41.65 W. However, because the function is nonlinear, the actual power at  $b = 13$  m is 1059.77 W (see Fig. 4.4). The relative derivative for



**Figure 4.4:** Power versus span and the corresponding derivative.

this same design can be computed as  $(\partial f / \partial b)(b/f) = -0.459$ , which means that for a 1% increase in span, the linear approximation predicts a 0.459% decrease in power; the actual decrease is 0.310%.

---

The gradient components quantify the rate of change of the function

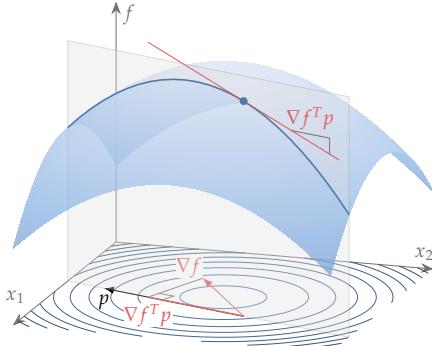
in each coordinate direction ( $x_i$ ), but sometimes we are interested in the rate of change in a direction that is not a coordinate direction. This corresponds to a *directional derivative*, which is defined as

$$\nabla_p f(x) \equiv \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon p) - f(x)}{\epsilon} \quad (4.7)$$

We can find this derivative by projecting the gradient onto the desired direction  $p$  using the dot product

$$\nabla_p f(x) = \nabla f^T p. \quad (4.8)$$

When  $p$  is a unit vector aligned with one of the Cartesian coordinates  $i$ , this dot product yields the corresponding partial derivative  $\partial f / \partial x_i$ . A two-dimensional example of this projection is shown in Fig. 4.5.



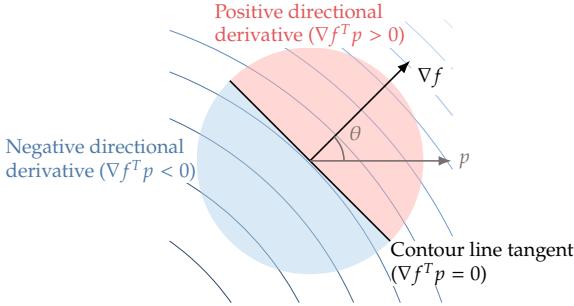
**Figure 4.5:** Projection of the gradient in an arbitrary unit direction  $p$ .

From the gradient projection, we can see why the gradient is the direction of steepest increase. If we use this definition of the dot product,

$$\nabla_p f(x) = \nabla f^T p = \|\nabla f\| \|p\| \cos \theta, \quad (4.9)$$

we can see that this is maximized when  $\theta = 0^\circ$ . That is, the directional derivative is largest when  $p$  points in the same direction as  $\nabla f$ . If  $-90^\circ < \theta < 90^\circ$ , the directional derivative is positive and is thus in a direction of increase (Fig. 4.6). If  $90^\circ < \theta < 270^\circ$ , the directional derivative is negative and  $p$  points in a descent direction. Finally, if  $\theta = \pm 90^\circ$ , the directional derivative is 0 and thus the function value does not change and is locally flat in that direction. That condition occurs if  $\nabla f$  and  $p$  are orthogonal, and thus the gradient is always orthogonal to contour surfaces.

To get the correct slope in the original units of  $x$ , the direction should be normalized as  $\hat{p} = p / \|p\|$ . In the gradient-based optimization



**Figure 4.6:** The gradient  $\nabla f$  is always orthogonal to contour lines (surfaces), and the directional derivative in the direction of  $p$  is given by  $\nabla f^T p$ .

algorithms of this chapter, it might not be necessary to normalize  $p$ . If  $p$  is not normalized, the slopes and variable axis are scaled by a constant.

**Example 4.3:** Directional derivative of a quadratic function

Consider the function of two variables,

$$f(x_1, x_2) = x_1^2 + 2x_2^2 - x_1x_2, \quad (4.10)$$

The gradient can be obtained using analytic differentiation to obtain

$$\nabla f(x_1, x_2) = \begin{bmatrix} 2x_1 - x_2 \\ 4x_2 - x_1 \end{bmatrix}. \quad (4.11)$$

At point  $x = [-1, 1]$ , the gradient is,

$$\nabla f(-1, 1) = \begin{bmatrix} -3 \\ 5 \end{bmatrix}. \quad (4.12)$$

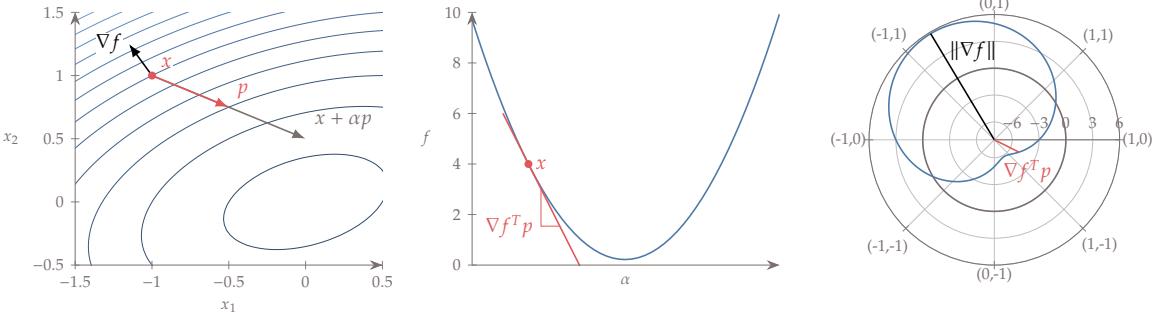
Taking the derivative in the direction  $p = [2/\sqrt{5}, -1/\sqrt{5}]^T$ , we obtain,

$$\nabla f^T p = [-3, 5] \begin{bmatrix} 2/\sqrt{5} \\ -1/\sqrt{5} \end{bmatrix} = -\frac{11}{\sqrt{5}}, \quad (4.13)$$

which we show in Fig. 4.7. We use a  $p$  with unit length to get the slope of the function in the original units. A projection of the function in the  $p$  direction can be obtained by plotting  $f$  along the line defined by  $x + \alpha p$ , where alpha is the independent variable, as shown in Ex. 4.3(middle). The projected slope of the function in that direction corresponds to the slope of this univariate function.

### 4.1.2 Curvature and Hessians

The rate of change of the gradient—the curvature—is also useful information because it tells us if a function slope is increasing (positive curvature), decreasing (negative curvature), or stationary (zero curvature).



In one dimension, the gradient reduces to a scalar (the slope) and the curvature is also a scalar that can be calculated by taking the second derivative of the function. To quantify curvature in  $n$  dimensions, we need to take the partial derivative of each gradient component  $j$  with respect to each coordinate direction  $i$ , which can be written as

$$\frac{\partial^2 f}{\partial x_i \partial x_j}. \quad (4.14)$$

If the function  $f$  has continuous second partial derivatives, the order of differentiation does not matter and the mixed partial derivatives are equal, and thus

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}. \quad (4.15)$$

This property is known as the symmetry of second derivatives or equality of mixed partials.\*

Considering all gradient components and their derivatives with respect to all coordinate directions results in a second-order tensor. This tensor can be represented as a square  $n \times n$  matrix of second-order partial derivatives called the *Hessian*:

$$\nabla(\nabla f(x)) \equiv H(x) \equiv \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \quad (4.16)$$

Because of the symmetry of second derivatives property, the Hessian is a symmetric matrix with  $n(n + 1)/2$  independent elements.

Each row  $i$  of the Hessian is a vector that quantifies rate of change of all components  $j$  of the gradient vector with respect to the  $i$  direction. On the other hand, each column  $j$  of the matrix quantifies the rate

**Figure 4.7:** Derivative along the direction  $p$ .

\*The symmetry of second derivatives has a long history and there is a lot more to it.

of change of component  $j$  of the gradient vector with respect to all coordinate directions  $i$ . Because the Hessian is symmetric, the rows and columns are transposes of each other and these two interpretations are equivalent.

We can find the rate of change of the gradient in an arbitrary direction  $p$  by taking the product  $Hp$ . This yields an  $n$ -vector that quantifies the rate of change in the gradient in the direction  $p$ , where each component of the vector is the rate of the change in the corresponding partial derivative with respect to a movement along  $p$ , so we can write as,

$$Hp = \nabla_p (\nabla f(x)) = \lim_{\epsilon \rightarrow 0} \frac{\nabla f(x + \epsilon p) - \nabla f(x)}{\epsilon}. \quad (4.17)$$

Because of the symmetry of second derivatives, we can also interpret this as the rate of change in the directional derivative of the function along  $p$  with respect to each of the components of  $p$ .

To find the curvature of the one-dimensional function along a direction  $p$ , we need to project  $Hp$  onto direction  $p$  as follows,

$$D_p^2 f(x) = p^T Hp, \quad (4.18)$$

which yields a scalar quantity. Again, if we want to get the curvature in the original units of  $x$ ,  $p$  should be normalized.

For an  $n$ -dimensional Hessian, it is possible to find  $n$  directions  $v$  along which projected curvature aligns with that direction, that is,

$$Hv = \kappa v. \quad (4.19)$$

This is an eigenvalue problem whose eigenvectors represent the *principal curvature* directions and the eigenvalues  $\kappa$  quantify the corresponding curvatures. If each eigenvector is normalized as  $\hat{v} = v/\|v\|_2$ , then the corresponding  $\kappa$  is the unscaled curvature.

---

**Example 4.4:** Hessian and principal curvature directions of a quadratic

Consider the quadratic function of two variables,

$$f(x_1, x_2) = x_1^2 + 2x_2^2 - x_1 x_2, \quad (4.20)$$

whose contours are shown in Fig. 4.8. These contours are ellipses that have the same focus points. The Hessian of this quadratic is

$$H = \begin{bmatrix} 2 & -1 \\ -1 & 4 \end{bmatrix}, \quad (4.21)$$

which is constant. To find the curvature in the direction  $p = [-1/2, -\sqrt{3}/2]^T$ , we compute

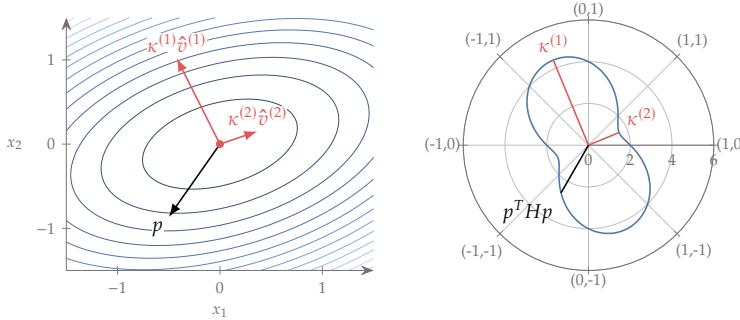
$$p^T Hp = \begin{bmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 4 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{bmatrix} = \frac{7 - \sqrt{3}}{2}. \quad (4.22)$$

The principal curvature directions can be computed by solving the eigenvalue problem (4.19). This yields two eigenvalues and two corresponding eigenvectors,

$$\kappa^{(1)} = 3 + \sqrt{2}, \quad v^{(1)} = \begin{bmatrix} 1 - \sqrt{2} \\ 1 \end{bmatrix}, \quad \text{and} \quad \kappa^{(2)} = 3 - \sqrt{2}, \quad v^{(2)} = \begin{bmatrix} 1 + \sqrt{2} \\ 1 \end{bmatrix}. \quad (4.23)$$

By plotting the principal curvature directions superimposed on the function contours (Fig. 4.8 on left), we can see that they are aligned with the major and minor axes of the ellipses.

To see how the curvature varies as a function of the direction, we make a polar plot of the curvature  $p^T H p$ , where  $p$  is normalized (Fig. 4.8 on right). We can see that the maximum curvature aligns with the first principal curvature direction, as expected, and the minimum one corresponds to the second principal curvature direction.



**Figure 4.8:** Contours of  $f$  for Ex. 4.4 and the two principal curvature directions in red. The polar plot shows the curvature, with the eigenvectors pointing at the directions of principal curvature, and all other directions have curvature values in between.

---

#### Example 4.5: Hessian of two-variable polynomial

Consider the same polynomial from Ex. 4.1. Differentiating the gradient we obtained previously, we obtain the Hessian,

$$H(x_1, x_2) = \begin{bmatrix} 6x_1 & 4x_2 \\ 4x_2 & 4x_1 - 6x_2 \end{bmatrix}. \quad (4.24)$$

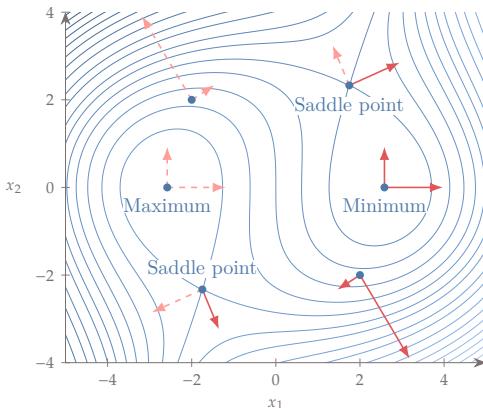
We can visualize the variation of the Hessian by plotting the principal curvatures at different points (Fig. 4.9).

---

### 4.1.3 Taylor Series

The Taylor series provides a local approximation to a function and is the foundation for gradient-based optimization algorithms.

For an  $n$ -dimensional function, the Taylor series can predict the function along any direction  $p$ . This is done by projecting the gradient



**Figure 4.9:** Principal curvature direction and magnitude variation.

and Hessian onto the desired direction  $p$ , to get an approximation of the function at any nearby point  $x + \alpha p$ : <sup>†</sup>

$$f(x + p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T H(x) p + O\left(\|p\|^3\right). \quad (4.25)$$

We use a second-order Taylor series (ignoring the  $\alpha^3$  term) because it results in a quadratic, which is the lowest order Taylor series that can have a minimum. For a function that is  $C^2$  continuous, this approximation can be made arbitrarily accurate by making  $\alpha$  small enough.

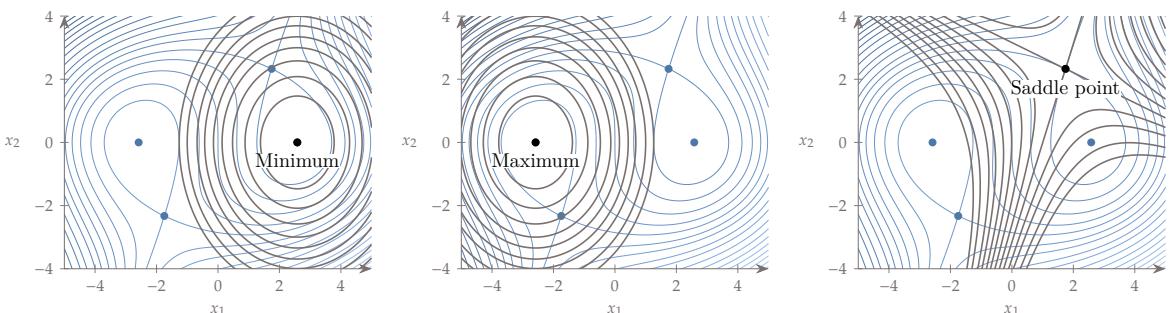
<sup>†</sup>For a more extensive introduction to the Taylor series, see Appendix A.6.

#### Example 4.6: Taylor series expansion of two-variable function

Using the gradient and Hessian of the two-variable polynomial from Ex. 4.1 and Ex. 4.5, we can use Eq. 4.25 to construct a second-order Taylor expansion about  $x^{(0)}$ ,

$$\tilde{f}(p) = f(x^{(0)}) + \begin{bmatrix} 3x_1^2 + 2x_2^2 - 20 \\ 4x_1 x_2 - 3x_2^2 \end{bmatrix}^T p + p^T \begin{bmatrix} 6x_1 & 4x_2 \\ 4x_2 & 4x_1 - 6x_2 \end{bmatrix} p. \quad (4.26)$$

Figure 4.10 shows the resulting Taylor series expansions about different points.



**Figure 4.10:** The second-order Taylor series expansion uses the function value, gradient, and Hessian at a point to construct a quadratic model about that point. Depending on the function, that point is the

#### 4.1.4 What is an Optimum?

To find the minimum of a function, we must determine the mathematical conditions that identify a given point  $x$  as a minimum. There is only a limited set of problems for which we can prove global optimality, so in general, we are only interested in local optimality.

The extreme value theorem states that a continuous function on a closed interval has both a maximum and a minimum in that interval. A point  $x^*$  is a local minimum if  $f(x^*) \leq f(x)$  for all  $x$  in the neighborhood of  $x^*$ . A second-order Taylor-series expansion about  $x^*$  for small steps of size  $p$  yields

$$f(x^* + p) = f(x^*) + \nabla f(x^*)^T p + \frac{1}{2} p^T H(x^*) p + \dots \quad (4.27)$$

For  $x^*$  to be an optimal point, we must have  $f(x^* + p) \geq f(x^*)$  for all  $p$ . This implies that the first- and second-order terms in the Taylor series have to be non-negative, that is,

$$\nabla f(x^*)^T p + \frac{1}{2} p^T H(x^*) p \geq 0. \quad (4.28)$$

Because the magnitude of  $p$  is small, we can always find a  $p$  such that the first term dominates. Therefore, we require that

$$\nabla f(x^*)^T p \geq 0. \quad (4.29)$$

Because  $p$  can be in any arbitrary direction, the only way this inequality can be satisfied is if all the elements of the gradient are zero,

$$\nabla f(x^*) = 0. \quad (4.30)$$

This is the *first-order optimality condition*. This is necessary because if any element of  $p$  is nonzero, there are directions (such as  $p = -\nabla f$ ) for which the inequality would not be satisfied.

Since the gradient term has to be zero, we must now satisfy the remaining term in the inequality (4.28), that is,

$$p^T H(x^*) p \geq 0. \quad (4.31)$$

From Eq. 4.18, we know that this term represents the curvature in direction  $p$ , so this means that the function curvature must be positive or zero when projected in any direction. You may recognize this

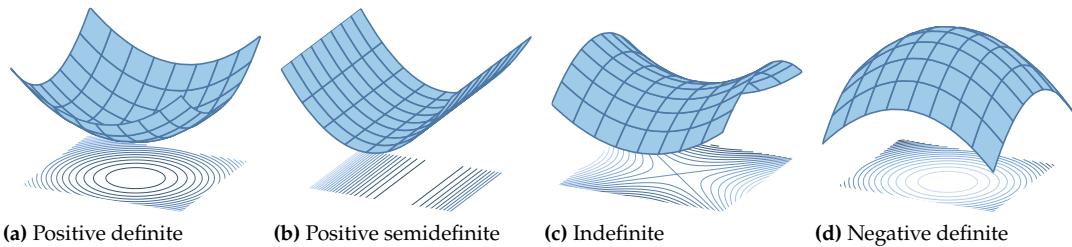
inequality as the definition of a *positive semidefinite* matrix. In other words, the Hessian  $H(x^*)$  must be positive semidefinite.

For a matrix to be positive semidefinite, its eigenvalues must all be greater than or equal to zero. Recall that the eigenvalues of the Hessian quantify the principal curvatures, so as long as all the principal curvatures are greater or equal than zero, the curvature along an arbitrary directions is also greater or equal than zero.

These conditions on the gradient and curvature are *necessary conditions* for a local minimum, but not sufficient. These conditions are not sufficient because if the curvature is zero in some direction  $p$  (i.e.,  $p^T H(x^*) p = 0$ ), we have no way of knowing if it is a minimum unless we look at the third-order term. In that case, even if it is a minimum, it is a weak minimum.

The *sufficient conditions* for optimality require that the curvature is positive in any direction, in which case we have a *strong minimum*. Mathematically, this means that  $p^T H(x^*) p > 0$  for all nonzero  $p$ , which is the definition of a *positive definite* matrix. If  $H$  is positive definite matrix, every eigenvalue of  $H$  is positive and the determinant of every leading principal sub-matrix of  $H$  is positive.

Figure 4.11 shows some examples of quadratic functions that are positive definite (all positive eigenvalues), positive semidefinite (non-negative eigenvalues), indefinite (mixed eigenvalues), and negative definite (all negative eigenvalues).



In summary, the *necessary optimality conditions* for an unconstrained optimization problem are

$$\begin{aligned} \nabla f(x^*) &= 0, \\ H(x^*) &\text{ is positive semidefinite.} \end{aligned} \tag{4.32}$$

The *sufficient optimality conditions* are

$$\begin{aligned} \nabla f(x^*) &= 0, \\ H(x^*) &\text{ is positive definite.} \end{aligned} \tag{4.33}$$

**Figure 4.11:** Quadratic functions with different types of Hessians from positive definite to negative definite.

---

**Example 4.7:** Finding minima analytically.

Consider the function to two variables,

$$f = 0.5x_1^4 + 2x_1^3 + 1.5x_1^2 + x_2^2 - 2x_1x_2$$

We can find the minima of this function analytically by solving for the optimality conditions analytically.

To find the critical points of this function, we solve for the points at which the gradient is equal to zero,

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1^3 + 6x_1^2 + 3x_1 - 2x_2 \\ 2x_2 - 2x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

From the second equation we have that  $x_2 = x_1$ . Substituting this into the first equation yields,

$$x_1(2x_1^2 + 6x_1 + 1) = 0.$$

The solutions of this equation yields three points

$$x^{(1)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad x^{(2)} = \begin{bmatrix} -\frac{3}{2} - \frac{\sqrt{7}}{2} \\ -\frac{3}{2} - \frac{\sqrt{7}}{2} \end{bmatrix}, \quad x^{(3)} = \begin{bmatrix} \frac{\sqrt{7}}{2} - \frac{3}{2} \\ \frac{\sqrt{7}}{2} - \frac{3}{2} \end{bmatrix}$$

To classify these points, we need to compute the Hessian matrix. Differentiating the gradient, we get

$$H(x_1, x_2) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 6x_1^2 + 12x_1 + 3 & -2 \\ -2 & 2 \end{bmatrix}.$$

One easy way to determine if a  $2 \times 2$  matrix is positive is by checking that both the first diagonal element and the matrix determinant are positive. Evaluating the determinant the first point, we get

$$\det(H(x^{(1)})) = \det \begin{bmatrix} 3 & -2 \\ -2 & 2 \end{bmatrix} = 2 > 0.$$

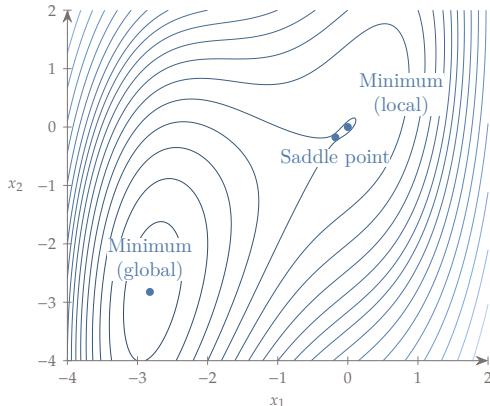
Since the first diagonal element is positive as well,  $H$  at this point is positive definite so  $x^{(1)}$  is a local minimum. For the second point,

$$\det(H(x^{(2)})) = \det \begin{bmatrix} 3(3 + \sqrt{7}) & -2 \\ -2 & 2 \end{bmatrix} = 14 + 6\sqrt{7} > 0.$$

Since  $3(3 + \sqrt{7}) > 0$  as well,  $x^{(2)}$  is also a local minimum. For the third point,

$$\det(H(x^{(3)})) = \det \begin{bmatrix} 9 - 3\sqrt{7} & -2 \\ -2 & 2 \end{bmatrix} = 14 - 6\sqrt{7} < 0,$$

and since  $9 - 3\sqrt{7} > 0$ , this is a saddle point.



**Figure 4.12:** Minima and critical points for a polynomial of two variables.

These three critical points are shown in Fig. 4.12. To find out which of the two local minima is the global one, we evaluate the function at each of these points. Since  $f(x^{(2)}) < f(x^{(1)})$ ,  $x^{(2)}$  is the global minimum.

While it is possible to solve for the optimality conditions analytically, as we did in Ex. 4.7, this is not possible in general because the resulting equations might not be solvable in closed form. Therefore, we need numerical methods for solving for these conditions.

When using a numerical approach, we seek points where  $\nabla f(x^*) = 0$ , but the entries in  $\nabla f$  do not converge to exactly zero because of finite-precision arithmetic. Instead, we define convergence for the first criterion based on the maximum component of the gradient, such that,

$$\|\nabla f\|_\infty < \tau, \quad (4.34)$$

where  $\tau$  is some tolerance. A typical absolute tolerance is  $\tau = 10^{-6}$  or a six-order magnitude reduction in gradient when using a relative tolerance. The second condition (that  $H$  must be positive semidefinite) is not usually checked explicitly. If we satisfy the first condition then all we know is that we have reached a stationary point, which could be a maximum, a minimum, or a saddle point. However, as shown in Section 4.4, our search directions are always descent directions and so in practice, we only converge to local minimum.

## 4.2 Two Overall Approaches to Finding an Optimum

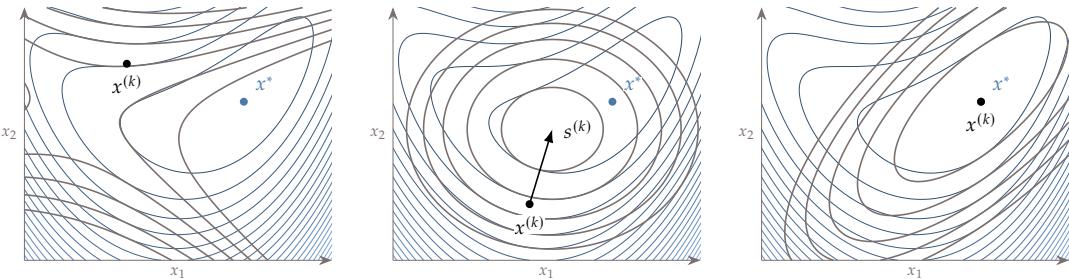
While the optimality conditions derived in the previous section can be solved analytically to find the function minima, this analytic approach is not possible for functions that are the results of numerical models.

Instead, we need iterative numerical methods that can find minima based only on the function values and its gradients.

In Chapter 3, we reviewed methods for solving simultaneous systems of nonlinear equations, which we wrote as  $r(u) = 0$ . Because the first order optimality condition ( $\nabla f = 0$ ) can be written in this residual form (where  $r \equiv \nabla f$  and  $u \equiv x$ ), we could try to use the solvers from Chapter 3 directly to solve unconstrained optimization problems. While several components of general solvers for  $r(u) = 0$  are used in optimization algorithms, these solvers are not the most effective approaches in their original form. Furthermore, solving  $\nabla f = 0$  is not necessarily sufficient—it finds a stationary point but not necessarily a minimum. Optimization algorithms require additional considerations to ensure convergence to a minimum.

Similarly to the iterative solvers from Chapter 3, gradient-based algorithms start with a guess,  $x^{(0)}$ , and generate a series of points,  $x^{(1)}, x^{(2)}, \dots, x^{(k)}, \dots$  that converge to a local optimum,  $x^*$ , as previously illustrated in Fig. 4.1. At each iteration, some form of the Taylor series about the current point is used to find the next point.

A truncated Taylor series is in general only a good model within a small neighborhood, as shown in Fig. 4.13, which shows three quadratic models of the same function based on three different points. All



quadratic approximations match the local gradient and curvature at the respective points. However, the Taylor series quadratic about the first point (left plot) yields a quadratic without a minimum (the only critical point is a saddle point). The second point (middle plot) yields a quadratic whose minimum is closer to the true minimum. Finally, the Taylor series about the actual minimum point (right plot) yields a quadratic with the same minimum, as would be expected, but we can see how the quadratic model worsens the further we are from the point.

Because the Taylor series is only guaranteed to be a good model locally, we need a *globalization* strategy to ensure convergence to an optimum. Globalization here means to make the algorithm robust

**Figure 4.13:** Taylor series quadratic models are only guaranteed to be accurate near the point about which the series is expanded ( $x$ ). When the point is far from the optimum, the quadratic model might result in a function without a minimum (left).

enough that it is able to converge to a local minimum starting from any point in the domain. This should not to be confused with trying to find the global minimum, which is a separate issue (see Tip 4.24). There are two main globalization strategies: line search and trust region.

The line search approach consists of three main steps for every iteration (Fig. 4.14):

1. Choose a suitable search direction from the current point. The choice of search direction is based on a Taylor series approximation.
2. Determine how far to move in that direction by performing a *line search*.
3. Move to the new point and update all values.

The two first steps can be seen as two separate subproblems. We address the line search subproblem in Section 4.3 and the search direction subproblem in Section 4.4.

Trust-region methods also consist of three steps (Fig. 4.15):

1. Create a model about the current point. This model can be based on a Taylor series approximation or another type of surrogate model.
2. Minimize the model within a *trust region* around the current point to find the step.
3. Move to the new point, update values, and adapt the size of the trust region.

We introduce the trust-region approach in Section 4.5, but we devote more attention to algorithms that use the line search approach because they are more common.

Both of the line search and trust region approaches use iterative processes that must be repeated until some convergence criterion is satisfied. The first step in both approaches is usually referred to as a *major* iteration, while the second step might require more function evaluations corresponding to *minor* iterations.

### 4.3 Line Search

All gradient-based unconstrained optimization algorithms that use a line search follow the procedure outlined in Alg. 4.8. We start with a guess  $x^{(0)}$  and provide a convergence tolerance  $\tau$  for the optimality condition. The final output is an optimal point  $x^*$  and the corresponding function value  $f(x^*)$ . As mentioned in the previous section, there are two main subproblems in line-search gradient-based optimization

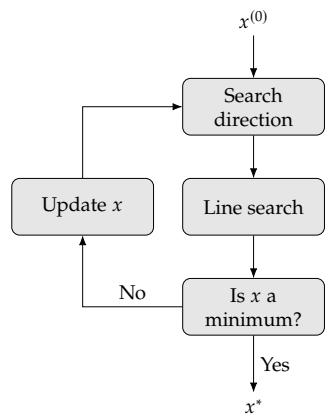


Figure 4.14: Line search approach.

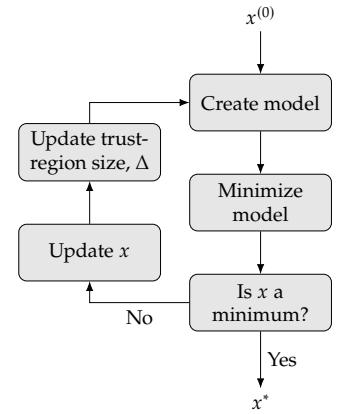


Figure 4.15: Trust region approach.

algorithms: choosing the search direction and determining how far to step in that direction. In the next section, we introduce several methods for choosing the search direction. The line search method determines how far to step in the chosen direction and is usually independent of the method for choosing the search direction. Therefore, different line search methods can be combined with different methods for finding the search direction. However, the search direction method determines the name of the overall optimization algorithm, as we will see in the next section.

---

**Algorithm 4.8:** Gradient-based unconstrained optimization using a line search

**Inputs:**

$x^{(0)}$ : Starting point

$\tau$ : Convergence tolerance

func: Function that takes  $x$  as input and returns  $f(x)$  and optionally  $\nabla f(x)$

**Outputs:**

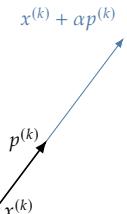
$x^*$ : Optimal point

$f(x^*)$ : Corresponding function value

---

<b>while</b> $\ \nabla f\ _\infty > \tau$ <b>do</b> Determine search direction, $p^{(k)}$ Determine step length, $\alpha^{(k)}$ $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$ $k = k + 1$ <b>end while</b>	<i>Optimality condition</i> <i>Use any of the methods from Section 4.4</i> <i>Use a line search algorithm</i> <i>Update design variables</i> <i>Increment iteration index</i>
--	---

---



For the line search subproblem, we assume that we are given a starting at  $x^{(k)}$  and a suitable search direction  $p^{(k)}$  along which we are going to search. The line search then operates solely on points along direction  $p^{(k)}$  starting from  $x^{(k)}$ , which can be written as

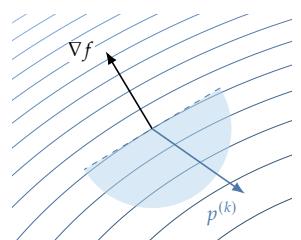
$$x^{(k+1)} = x^{(k)} + \alpha p^{(k)}, \quad (4.35)$$

where the scalar  $\alpha$  is always positive and represents how far we go in the direction  $p^{(k)}$ . This equation produces a one-dimensional slice of  $n$ -dimensional space, as illustrated in Fig. 4.17.

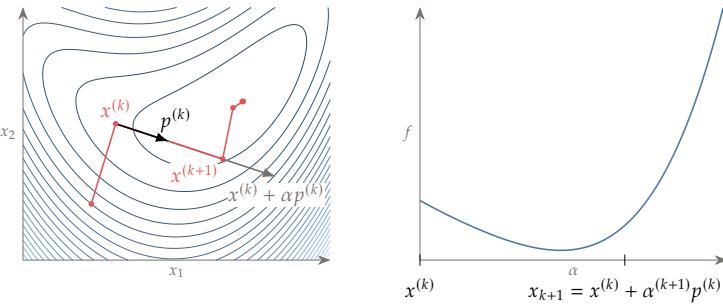
The line search determines the magnitude of the scalar  $\alpha^{(k)}$ , which in turn determines the next point in the iteration sequence. Even though  $x^{(k)}$  and  $p^{(k)}$  are  $n$ -dimensional, the line search is a one-dimensional problem with the goal of selecting  $\alpha^{(k)}$ .

Line search methods require that the search direction  $p^{(k)}$  be a *descent direction*, so that  $\nabla f^{(k)}^T p^{(k)} < 0$  (see Fig. 4.6). This guarantees

**Figure 4.16:** The line search starts from a given point  $x^{(k)}$  and searches solely along direction  $p^{(k)}$ .



**Figure 4.18:** The line search direction must be a descent direction.



**Figure 4.17:** The line search projects the  $n$ -dimensional problem onto one-dimension, where the independent variable is  $\alpha$ .

that  $f$  can be reduced by stepping some distance along this direction with a positive  $\alpha$ .

The goal of the line search is *not* to find the value of  $\alpha^{(k)}$  that minimizes  $f(x^{(k)} + \alpha^{(k)} p^{(k)})$ , but to find a point that is “good enough” using as few function evaluations as possible. This is because finding the exact minimum along the line would require too many evaluations of the objective function and possibly its gradient. Because the overall optimization needs to find a point in  $n$ -dimensional space, the search direction might change drastically between line searches, so spending too many iterations on each line search is generally not worthwhile.

Consider the function shown in Fig. 4.19. At point  $x^{(k)}$ , the direction  $p^{(k)}$  is a descent direction. However, it would be wasteful to spend a lot of effort determining the exact minimum in the  $p^{(k)}$  direction because it would not take us any closer to the minimum of the overall function (the dot on the right side of the plot). Instead, we should find a point that is good enough and then update the search direction.

To simplify the notation for the line search, we define the univariate function

$$\phi(\alpha) = f\left(x^{(k)} + \alpha p^{(k)}\right), \quad (4.36)$$

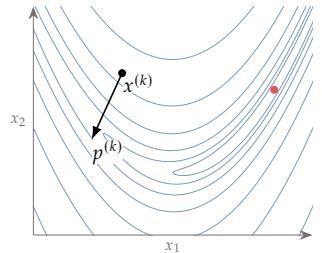
where  $\alpha = 0$  corresponds to the start of the line search and thus  $\phi(0) = f(x^{(k)})$ . Then, using  $x = x^{(k)} + \alpha p^{(k)}$ , the slope of the univariate function is

$$\begin{aligned} \phi'(\alpha) &= \frac{\partial[f(x)]}{\partial\alpha} = \frac{\partial[f(x)]}{\partial x} \frac{\partial x}{\partial\alpha} = \nabla f(x)^T p^{(k)} \\ &= \nabla f\left(x^{(k)} + \alpha p^{(k)}\right)^T p^{(k)}, \end{aligned} \quad (4.37)$$

which is the directional derivative along the search direction. The slope at the start of a given line search is

$$\phi'(0) = \nabla f^{(k)}^T p^{(k)}. \quad (4.38)$$

Because  $p^{(k)}$  must be a descent direction,  $\phi'(0)$  is always negative. Fig. 4.20 is a version of the one-dimensional slice from Fig. 4.17 in the



**Figure 4.19:** The descent direction does not necessarily point towards the minimum, in which case it would be wasteful to do an exact line search.

this notation. The  $\alpha$  axis and the slopes scale with the magnitude of  $p^{(k)}$ .

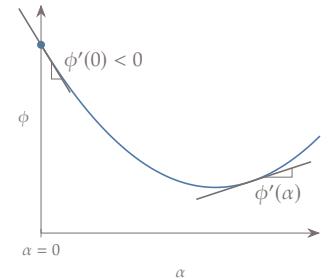
### 4.3.1 Sufficient Decrease and Backtracking

The simplest line search algorithm to find a “good enough” point relies on the *sufficient decrease condition* in combination with a *backtracking algorithm*. The sufficient decrease condition, also known as the *Armijo condition*, is given by the inequality

$$\phi(\alpha) \leq \phi(0) + \mu_1 \alpha \phi'(0) \quad (4.39)$$

for a constant  $0 < \mu_1 \leq 1$ .<sup>‡</sup> The quantity  $\alpha \phi'(0)$  represents the expected decrease of the function, assuming the function continued at the same slope. The multiplier  $\mu_1$  states that we will be satisfied as long we achieve even a small fraction of the expected decrease. In practice, this constant is several orders of magnitude smaller than one, typically  $\mu_1 = 10^{-4}$ . Because  $p^{(k)}$  is a descent direction, and thus  $\phi'(0) = \nabla f^{(k)} T p^{(k)} < 0$ , there is always a positive  $\alpha$  that satisfies this condition for a smooth function.

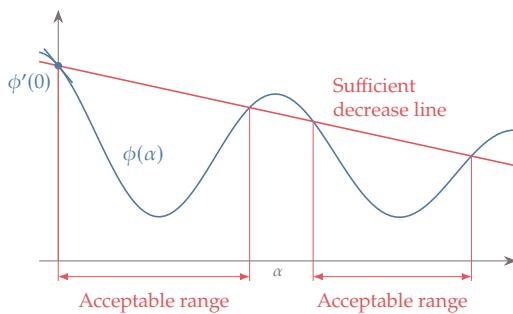
The concept is illustrated in Fig. 4.21, which shows a function with negative slope at  $\alpha = 0$  and a sufficient decrease line whose slope is a fraction of that initial slope. When starting a line search, all we know is the function value and its slope at  $\alpha = 0$ , so we do not really know how the function varies until we evaluate it. Because we do not want to do too many function evaluations, the first point whose value is below the sufficient decrease line is deemed acceptable. The sufficient decrease line slope in Fig. 4.21 is exaggerated for illustration purposes; for typical values of  $\mu_1$ , the line is indistinguishable from a horizontal line when plotted.



**Figure 4.20:** For the line search, we denote the function as  $\phi(\alpha)$  with the same value as  $f$ . The slope  $\phi'(\alpha)$  is the gradient of  $f$  projected onto the search direction.

<sup>‡</sup>This condition can be problematic near a local minimum because  $\phi(0)$  and  $\phi(\alpha)$  are very similar and so their subtraction is inaccurate. Hager *et al.*<sup>55</sup> introduced an approximate Wolfe condition with improved accuracy along with an efficient line search based on a secant method.

55. Hager *et al.*, *A New Conjugate Gradient Method with Guaranteed Descent and an Efficient Line Search*. 2005



**Figure 4.21:** Sufficient decrease conditions.

Line search algorithms require a first guess for  $\alpha$ . As we will see later, some methods for finding the search direction also provide good

guesses for the step length. However, in many cases we have no idea of the scale of function, so our initial guess may not be suitable. Even if we do have an educated guess for  $\alpha$ , it is only a guess and the first step might not satisfy the sufficient decrease condition.

One simple algorithm that is guaranteed to find a step that satisfies the sufficient decrease condition is backtracking (Alg. 4.9). This algorithm starts with a maximum step and successively reduces the step by a constant ratio  $\rho$  until it satisfies the sufficient decrease condition (a typical value is  $\rho = 0.5$ ). Because our search direction is a descent direction, we know that if we backtrack enough we will achieve an acceptable decrease in function value.

Algorithm 4.9: Backtracking line search algorithm

## Inputs:

- $\alpha_{\text{init}} > 0$ : Initial step length
- $0 < \mu_1 < 1$ : Sufficient decrease factor
- $0 < \rho < 1$ : Backtracking factor

## Outputs:

$\alpha^*$ : Step size satisfying sufficient decrease condition

```

 $\alpha = \alpha_{\text{init}}$ 
while  $\phi(\alpha) > \phi(0) + \mu_1 \alpha \phi'(0)$  do Is function value is above sufficient decrease line?
     $\alpha = \rho \alpha$ 
end while Backtrack

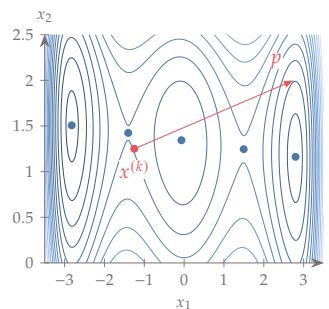
```

### Example 4.10: Backtracking line search

Consider the function,

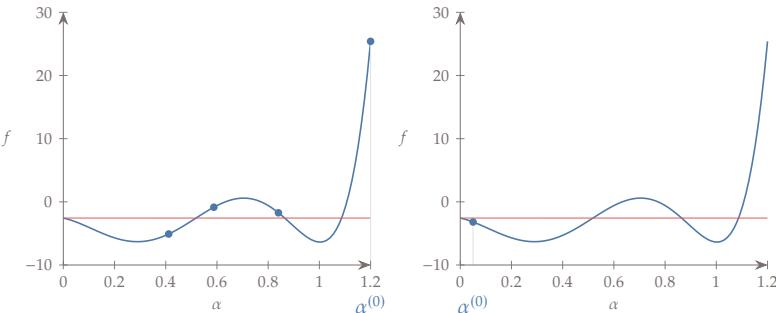
$$f(x_1, x_2) = 0.1x_1^6 - 1.5x_1^4 + 5x_1^2 + 0.1x_2^4 + 3x_2^2 - 9x_2 + 0.5x_1x_2.$$

Suppose we do a line search starting from  $x = (-1.25, 1.25)$  in the direction  $p = [4, 0.75]$ , as shown in Fig. 4.22. Applying the backtracking algorithm with  $\mu_1 = 10^{-4}$  and  $\rho = 0.65$  produces the iterations shown in Fig. 4.23. The sufficient decrease line appears to be horizontal, but it just has a small slope because  $\mu_1$  is small. Using a large initial step of  $\alpha_{\text{init}} = 1.2$  (left), several iterations are required. For a small initial step of  $\alpha_{\text{init}} = 0.05$  (right), the algorithm satisfies sufficient decrease at the first iteration but misses out on further decreases.



**Figure 4.22:** Line search direction.

Although backtracking is guaranteed to find a point that satisfies sufficient decrease, there are two undesirable scenarios where this algorithm performs poorly. The first scenario is that the guess for the



**Figure 4.23:** Backtracking using different initial steps.

initial step is far too large, and the step sizes that satisfy sufficient decrease are smaller than the starting step by several orders of magnitude. Depending on the value of  $\rho$ , this scenario requires a large number of backtracking evaluations.

The other undesirable scenario is where our initial guess immediately satisfies sufficient decrease, but the slope of the function at this point is still highly negative and we could have decreased the function value by much more if we had taken a larger step. In this case, our guess for the initial step is far too small.

Even if our original step size is not too far from an acceptable step size, the basic backtracking algorithm ignores any information we have about the function values and its gradients, and blindly takes a reduced step based on a preselected ratio  $\rho$ . We can make more intelligent estimates of where an acceptable step is based on the evaluated function values (and gradients, if available). In the next section, we introduce a more sophisticated line search algorithm that is able to deal with these scenarios much more efficiently.

### 4.3.2 A Better Line Search

One major weakness of the sufficient decrease condition is that it accepts small steps that marginally decrease the objective function, because  $\mu_1$  in Eq. 4.39 is rather small. We could just increase  $\mu_1$  (that is, tilt the red line downward in Fig. 4.21) to prevent these small steps; however, that would prevent us from taking large steps that result in a reasonable decrease. A large step that provides a reasonable decrease is desirable, because that progress generally leads to faster convergence. Instead, we want to prevent overly small steps while not making it more difficult to accept decent large steps. This is accomplished by adding a second condition and using it to construct a more efficient line search algorithm.

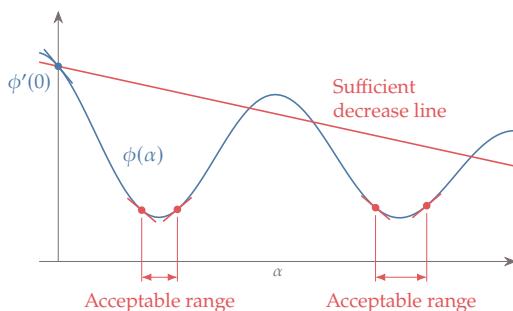
Just like guessing the step size, it is difficult to know in advance how much of a function value decrease to expect. However, if we compare

the slope of the function at the candidate point with the slope at the start of the line search, we can get an idea if the function is “bottoming out,” or flattening, using the *curvature condition*:

$$|\phi'(\alpha)| \leq \mu_2 |\phi'(0)|. \quad (4.40)$$

This condition requires that the magnitude of the slope at the new point be lower than the magnitude of the slope at the start of the line search by a factor of  $\mu_2$ . This requirement is called the curvature condition because by comparing the two slopes, we are effectively quantifying the curvature of the function. Typical values of  $\mu_2$  range from 0.1 to 0.9, and the best value depends on the method for determining the search direction and is also problem dependent. To guarantee that there are steps that satisfy both sufficient decrease and sufficient curvature, the sufficient decrease slope must be shallower than the sufficient curvature slope, that is,  $0 < \mu_1 \leq \mu_2 \leq 1$ . As  $\mu_2$  tends to zero, enforcing the sufficient curvature condition tends toward an exact line search.

The sign of the slope at a point satisfying this condition is not important; all that matters is that the function be shallow enough. The idea is that if the slope  $\phi'(\alpha)$  is still negative with a magnitude similar to the slope at the start of the line search, then the step is too small, and we expect the function to decrease even further by taking a larger step. If the slope  $\phi'(\alpha)$  is positive with a magnitude similar to that at the start of the line search, then the step is too large, and we expect to decrease the function further by taking a smaller step. On the other hand, when the slope is shallow enough (either positive or negative), we assume that the candidate point is near a local minimum and additional effort will yield only incremental benefits that are wasteful in the context of our larger problem. The sufficient decrease and curvature conditions are collectively known as the *strong Wolfe conditions*. Figure 4.24 shows acceptable intervals that satisfy the strong Wolfe conditions.



**Figure 4.24:** Steps that satisfy the strong Wolfe conditions.

We now develop a more efficient line search algorithm that finds

a step satisfying the strong Wolfe conditions. Note that using the curvature condition means we require derivative information ( $\phi'$ ). There are various line search algorithms in the literature, including some that are derivative-free. Here, we detail a line search algorithm similar to that presented by Nocedal and Wright<sup>56</sup>. The algorithm has two stages:

1. The *bracketing* stage finds an interval within which we are certain to find an acceptable step.
2. The *pinpointing* stage finds a point that satisfies the strong Wolfe conditions within the interval provided by the bracketing stage.

56. Nocedal et al., *Numerical Optimization*. 2006

The bracketing stage is detailed in Alg. 4.11 and illustrated in Fig. 4.25; it consists of increasing the step size until finding a step that satisfies the strong Wolfe conditions, or until finding an interval that must contain a point satisfying those conditions. For a smooth continuous function, the conditions are guaranteed to be met by a point in a given interval if:

1. The function value at the candidate step is higher than at the start of the line search.
2. The step satisfies sufficient decrease, but the slope is positive.

If the step satisfies sufficient decrease and the slope is negative, the step size is increased to look for a larger function value reduction along the line.

---

**Algorithm 4.11:** Bracketing stage for the line search algorithm

---

**Inputs:**

$\alpha_1 > 0$ : *Initial step size guess*  
 $0 < \mu_1 < 1$ : *Sufficient decrease factor*  
 $0 < \mu_2 < 1$ : *Sufficient curvature factor*  
 $\rho > 1$ : *Step size increase factor*

**Outputs:**

$\alpha^*$ : *Step size satisfying strong Wolfe conditions*

---

```

 $\alpha_0 = 0$ 
 $i = 1$ 
while true do
    Evaluate  $\phi(\alpha_i)$ 
    if [ $\phi(\alpha_i) > \phi(0) + \mu_1 \alpha_i \phi'(0)$ ] or [ $\phi(\alpha_i) > \phi(\alpha_{i-1})$  and  $i > 1$ ] then
         $\alpha^* = \text{pinpoint}(\alpha_{i-1}, \alpha_i)$  return  $\alpha^*$ 
    end if

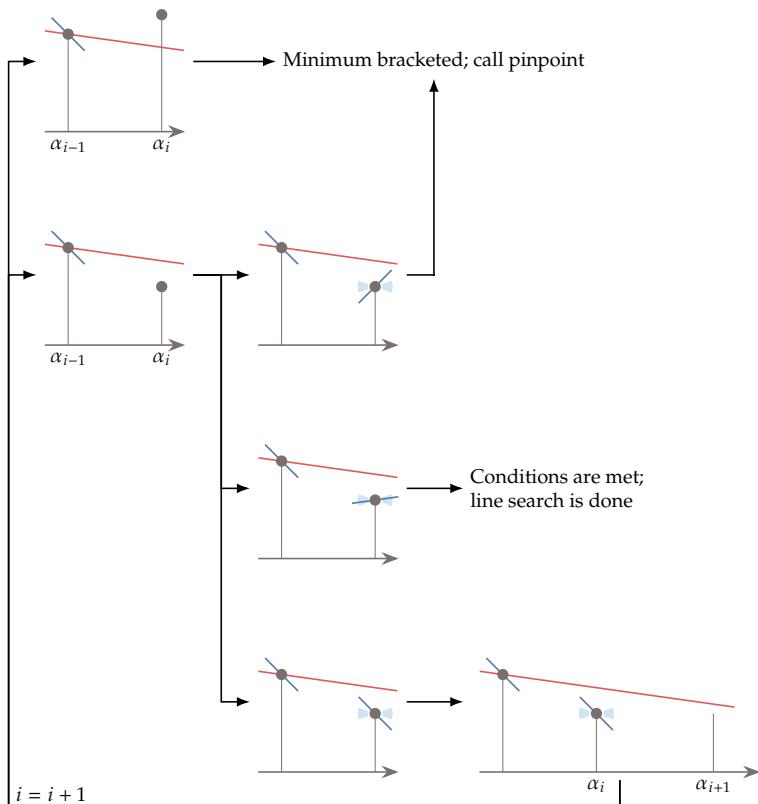
```

```

Evaluate  $\phi'(\alpha_i)$ 
if  $|\phi'(\alpha_i)| \leq -\mu_2 \phi'(0)$  then return  $\alpha^* = \alpha_i$ 
else if  $\phi'(\alpha_i) \geq 0$  then
     $\alpha^* = \text{pinpoint}(\alpha_i, \alpha_{i-1})$  return  $\alpha^*$ 
else
     $\alpha_{i+1} = \rho \alpha_i$ 
    end if
     $i = i + 1$ 
end while

```

---



**Figure 4.25:** Visual representation of the bracketing algorithm.

The algorithm for the second stage, the  $\text{pinpoint}(\alpha_{\text{low}}, \alpha_{\text{high}})$  function, is given in Alg. 4.12. In the first step, we need to estimate a good candidate point within the interval that is expected to satisfy the strong Wolfe conditions. A number of algorithms can be used to find such a point. Since we have the function value and derivative at one endpoint of the interval, and at least the function value at the other endpoint, one

option is to perform quadratic interpolation to estimate the minimum within the interval. If the two end points are  $\alpha_1$  and  $\alpha_2$ , respectively, the minimum can be found analytically from the function values and derivative as

$$\alpha_{\min} = \frac{2\alpha_1 [\phi(\alpha_2) - \phi(\alpha_1)] + \phi'(\alpha_1)(\alpha_1^2 - \alpha_2^2)}{2 [\phi(\alpha_2) - \phi(\alpha_1) + \phi'(\alpha_1)(\alpha_1 - \alpha_2)]}. \quad (4.41)$$

If we provide analytic gradients, or we already evaluated  $\phi'(\alpha_i)$  (either as part of Alg. 4.11 or as part of checking the strong Wolfe conditions in Alg. 4.12), then we would have the function values and derivatives at both points and we could use cubic interpolation instead.

A graphical representation of the process is shown in Fig. 4.26. In the leftmost figure we construct a quadratic fit based on the function value and slope at  $\alpha_{\text{low}}$  and the function value at  $\alpha_{\text{high}}$ . This fit provides us with a good estimate of where the minimum might be. Four scenarios are possible for this new trial point. In the first three the function value is too high, the slope is too positive, or the slope is too negative. In those scenarios we update our bracket and restart. In the fourth scenario the function value decreases sufficiently, and the slope is sufficiently small in magnitude to satisfy the strong Wolfe conditions.

---

**Algorithm 4.12:** Pinpoint function for the line search algorithm
 

---

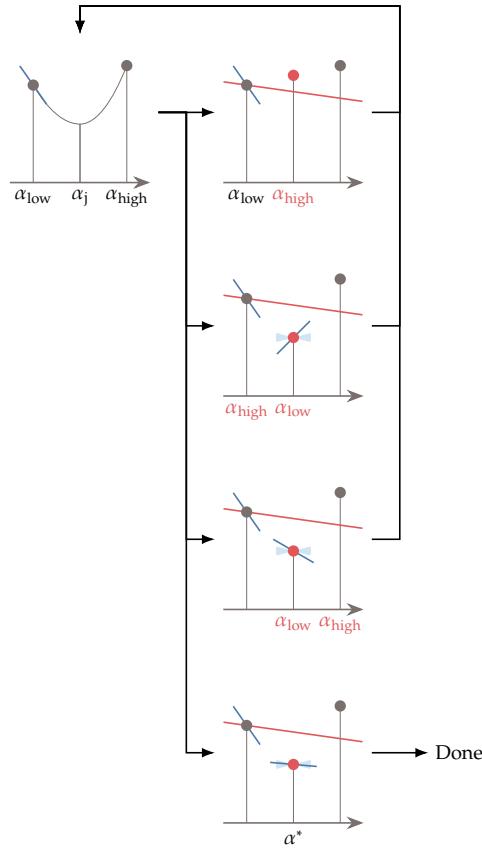
**Inputs:** $\alpha_{\text{low}}$ : Lower limit for pinpoint function $\alpha_{\text{high}}$ : Upper limit for pinpoint function $0 < \mu_1 < 1$ : Sufficient decrease factor $0 < \mu_2 < 1$ : Sufficient curvature factor**Outputs:** $\alpha^*$ : Step size satisfying strong Wolfe conditions $j = 0$ **while** true **do**Find  $\alpha_{\text{low}} \leq \alpha_j \leq \alpha_{\text{high}}$  Using quadratic (4.41) or cubic interpolationEvaluate  $\phi(\alpha_j)$ **if**  $\phi(\alpha_j) > \phi(0) + \mu_1 \alpha_j \phi'(0)$  or  $\phi(\alpha_j) > \phi(\alpha_{\text{low}})$  **then**     $\alpha_{\text{high}} = \alpha_j$ **else**    Evaluate  $\phi''(\alpha_j)$     **if**  $|\phi'(\alpha_j)| \leq -\mu_2 \phi'(0)$  **then**         $\alpha^* = \alpha_j$  **return**  $\alpha^*$     **else if**  $\phi'(\alpha_j)(\alpha_{\text{high}} - \alpha_{\text{low}}) \geq 0$  **then**         $\alpha_{\text{high}} = \alpha_{\text{low}}$     **end if**

```

 $\alpha_{\text{low}} = \alpha_j$ 
end if
 $j = j + 1$ 
end while

```

---

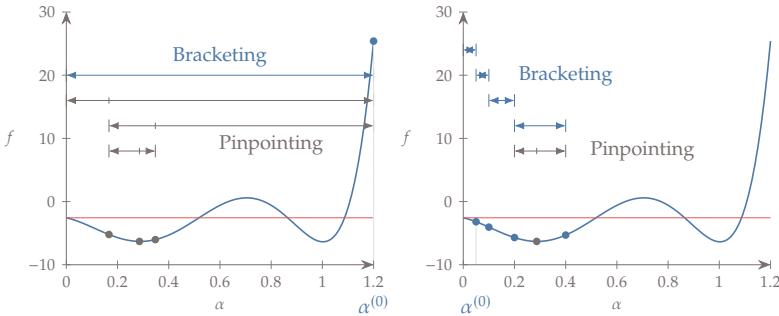


**Figure 4.26:** Visual representation of the pinpointing algorithm.

The line search defined by Alg. 4.11 followed by Alg. 4.12 is guaranteed to find a step length satisfying the strong Wolfe conditions for any parameters  $\mu_1$  and  $\mu_2$ . A robust algorithm needs to consider additional issues. One of these criteria is to ensure that the new point in the pinpoint algorithm is not so close to an endpoint as to cause the interpolation to be ill conditioned. A fall-back option in case the interpolation fails could be a simpler algorithm, such as bisection. Another of these criteria is to ensure that the loop does not continue indefinitely because finite-precision arithmetic leads to indistinguishable function

value changes.

**Example 4.13:** Line search with bracketing and pinpointing.



**Figure 4.27:** Example of a line search iteration.

Let us perform the same line search as in Alg. 4.9 but now using bracketing and pinpointing instead of backtracking. Using a large initial step of  $\alpha_{\text{init}} = 1.2$  (left), bracketing is achieved in the first iteration. Then pinpointing finds a point better than the one found using backtracking. The small initial step of  $\alpha_{\text{init}} = 0.05$  (right) does not satisfy the strong Wolfe conditions and the bracketing stage moves forward as long as the function keeps decreasing. The end result is a point that is much better than the one obtained with backtracking.

## 4.4 Search Direction

As stated in the beginning of this chapter, each iteration of an unconstrained gradient-based algorithm consists of two main steps: determining the search direction, and performing the line search (Alg. 4.8). The method used to find the search direction,  $p^{(k)}$ , in this iteration is what names the particular algorithm, which can use any of the line search algorithms described in the previous section. We start by introducing two first-order methods that only require the gradient and then explain two second-order methods that require the Hessian, or at least an approximation of the Hessian.

### 4.4.1 Steepest Descent

The steepest descent method (often called gradient descent) is a simple and intuitive method for determining the search direction. As discussed in Section 4.1.1, the gradient points in the direction of steepest increase, so  $-\nabla f$  points in the direction of steepest descent. Thus our search

direction at iteration  $k$  is simply

$$p = -\nabla f, \quad (4.42)$$

or as a normalized direction

$$p^{(k)} = -\frac{\nabla f^{(k)}}{\|\nabla f^{(k)}\|}. \quad (4.43)$$

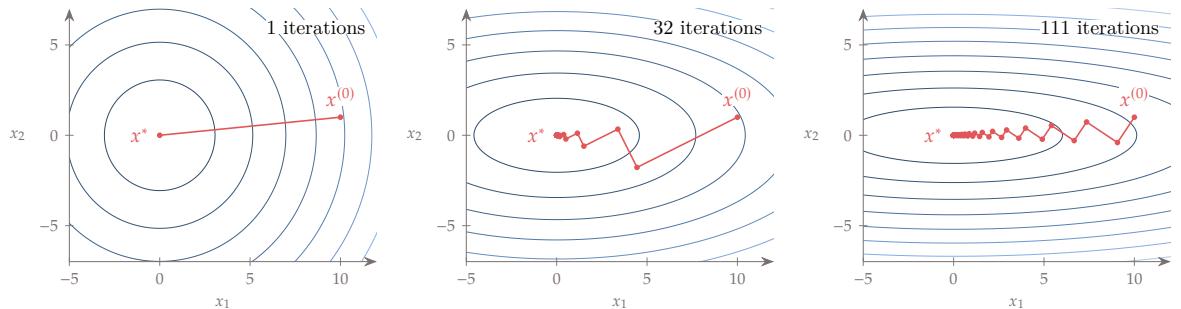
While steepest descent sounds like the best possible search direction to decrease a function, it actually is not. The reason is that when a function curvature varies greatly with direction, the gradient alone is a poor representation of function behavior beyond a small neighborhood.

**Example 4.14:** Steepest descent with varying amount of curvature

Consider the quadratic function,

$$f(x_1, x_2) = x_1^2 + \beta x_2^2,$$

where  $\beta$  can be set to adjust the curvature in the  $x_2$  direction. In Fig. 4.28, we show this function for  $\beta = 1, 5, 15$ . The starting point is  $x^{(0)} = (10, 1)$ . When



$\beta = 1$  (left), this quadratic has the same curvature in all directions and the steepest descent direction points directly to the minimum. When  $\beta > 1$  (middle and right), this is no longer the case and steepest descent shows abrupt changes in the subsequent search directions. This zigzagging is an inefficient way to approach the minimum. The higher the difference in curvature, the more iterations it takes.

**Figure 4.28:** Iteration history for a quadratic function using the steepest descent method with an exact line search.

The behavior shown in Ex. 4.14 is expected and we can show it mathematically. Assuming we perform an exact line search at each iteration, this means selecting the optimal value for  $\alpha$  along the line

search:

$$\begin{aligned} \frac{\partial f(x^{(k)} + \alpha p^{(k)})}{\partial \alpha} &= 0 \\ \frac{\partial f(x^{(k+1)})}{\partial \alpha} &= 0 \\ \frac{\partial f(x^{(k+1)})}{\partial x^{(k+1)}} \frac{\partial(x^{(k)} + \alpha p^{(k)})}{\partial \alpha} &= 0 \quad (4.44) \\ \nabla f^{(k+1)}^T p^{(k)} &= 0 \\ -p^{(k+1)}^T p^{(k)} &= 0 \end{aligned}$$

Hence each search direction is orthogonal to the previous one. As discussed in the last section, exact line searches are not desirable, so the search directions are not precisely orthogonal. However, the overall zigzagging behavior still exists.

Another issue with steepest descent is that the gradient at the current point on its own does not provide enough information to inform a good guess of the initial step size. As we saw in the line search, this initial choice has a large impact on the efficiency of the line search because the first guess could be orders of magnitude too small or too large. Second-order methods later in this section will help with this problem. In the meantime we can make a guess of the step size for a given line search based on the result of the previous one. If we assume that at the current line search we will obtain a decrease in objective function that is comparable to the previous one, we can write

$$\alpha^{(k)} \nabla f^{(k)}^T p^{(k)} \approx \alpha^{(k-1)} \nabla f^{(k-1)}^T p^{(k-1)}. \quad (4.45)$$

Solving for the step length, and inserting the steepest descent direction, we get the guess

$$\alpha^{(k)} = \alpha^{(k-1)} \frac{\|\nabla f^{(k-1)}\|^2}{\|\nabla f^{(k)}\|^2}. \quad (4.46)$$

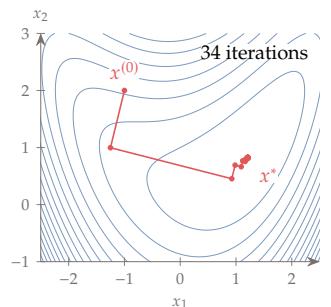
This is just the first guess in the new line search, which will then proceed as usual. If the slope of the function decreases relative to the previous line search, this guess decreases relative to the previous line search step length, and vice versa.

#### Example 4.15: Steepest descent applied to the bean function

We now find the minimum of the *bean function*,

$$f(x_1, x_2) = (1 - x_1)^2 + (1 - x_2)^2 + \frac{1}{2} (2x_2 - x_1^2)^2, \quad (4.47)$$

using the steepest descent algorithm with a two-stage line search. Using an exact



**Figure 4.29:** Steepest descent optimization path.

line search (small enough  $\mu_2$ ) and a convergence tolerance to  $\|\nabla f\|_\infty \leq 10^{-6}$ . The optimization path is shown in Fig. 4.29. While it takes only a few iterations to get close to the minimum, it takes many more to satisfy the specified convergence tolerance.

---

**Tip 4.16:** Problem scaling is important.

Problem scaling is one of the most important practical considerations in optimization. Steepest descent is particularly sensitive to scaling. Even though we will see methods that are less sensitive, for general nonlinear functions poor scaling can decrease the effectiveness of any method.

A common cause of poor scaling is unit choice. For example, consider a problem with two types of design variables, where one type is the material thickness in the order of  $10^{-6}$  m, and the other type is the length of the structure in the order of 1 m. If both distances are measured in meters, then the derivative in the thickness direction will be large compared to the derivative in the length direction. In other words, the design space will have a valley that is extremely steep and short in one direction, and gradual and long in the other. The optimizer will have great difficulty in navigating this type of design space.

Similarly, if the objective was power and a typical value was 1,000,000 W then all of the gradients will likely be relatively small and satisfying convergence tolerances may be difficult.

A good starting point for many optimization problems is to scale the objective and every design variable to be around unity. So in the first example we might measure thicknesses in micrometers, and in the second example we could report power in MW. This heuristic still does not guarantee that the derivatives are well scaled but it often provides a reasonable starting point for further fine tuning of the problem scaling.

---

#### 4.4.2 Conjugate Gradient

Steepest descent generally performs quite poorly, especially if the problem is not well scaled, like the quadratic example in Figure 4.28. The conjugate gradient method corrects the search directions such that they do not zigzag as much. This method is based on the linear conjugate gradient method, which was designed to solve linear equations. We first introduce the linear conjugate gradient method, and then adapt it to the nonlinear case.

For the moment, let us assume that we have a quadratic objective function. The Hessian of a quadratic function that is badly scaled has a high condition number, while a quadratic with a condition number of 1 is well scaled and would have perfectly circular contours. Fortunately, a

simple change in the search directions can yield a dramatic improvement for the badly scaled case. We can choose search directions that are less sensitive to the problem scaling. Let us also start with the simplest case, where the Hessian is the identity matrix. In that case, the function contours would all be circular. If we pick the coordinate directions as search directions, and find the minimum in each direction, then we can reach the minimum of an  $n$ -dimensional quadratic in at most  $n$  steps (Fig. 4.30).

Let us now assume that our quadratic is not ideally scaled, but rather stretched in some direction (and optionally rotated). We want to use the same concept and choose directions that will get us to the minimum in at most  $n$  steps. In the previous case, we chose orthogonal vectors as our search directions. For the more general case, we chose conjugate vectors. You can think of conjugate vectors as a generalization of orthogonal vectors. The vectors  $p$  are conjugate with respect to the Hessian (or  $H$ -orthogonal) if

$$p_i^T H p_j = 0 \text{ for all } i \neq j. \quad (4.48)$$

If the Hessian is the identity matrix, this definition would produce an orthogonal set of vectors. Conjugate vectors are “orthogonal” in the stretched sense. These conjugate directions retain the property that if we determine the best step in each conjugate direction, we can reach the minimum of an  $n$ -dimensional quadratic in  $n$  steps (Fig. 4.31). This is a significant improvement over steepest descent, which, as we have seen, can take many iterations to converge on a stretched two-dimensional quadratic function.

Of course a general function is not quadratic, and our line search methods do not find the best step in each direction. However, we address these issues by using a local quadratic approximation of the function, and performing a periodic restart, where every  $n$  iterations a steepest descent step is taken instead.

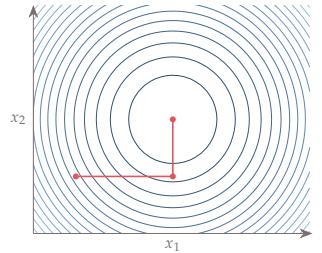
In practice this method outperforms steepest descent significantly with only a small modification in procedure. The required change is to save information on the search direction and gradient from the previous iteration:

$$p^{(k)} = -\nabla f^{(k)} + \beta^{(k)} p^{(k-1)}, \quad (4.49)$$

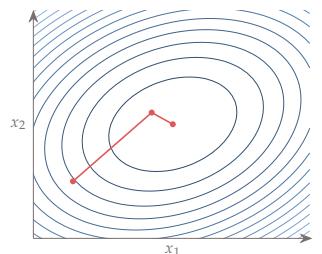
where

$$\beta^{(k)} = \frac{\nabla f^{(k)T} \nabla f^{(k)}}{\nabla f^{(k-1)T} \nabla f^{(k-1)}}. \quad (4.50)$$

The parameter  $\beta$  can be interpreted as a “damping parameter” that prevents each search direction from varying too much relative to the



**Figure 4.30:** For a quadratic function with perfectly circular contours (condition number of 1) we can find the minimum in  $n$  steps, where  $n$  is the number of dimensions, by using a coordinate search. A coordinate search just means that we find that minimum in each coordinate sequentially. So in the above example we searched in the  $x_1$  direction to find the minimum along that line, then searched in the  $x_2$  direction.



**Figure 4.31:** For any quadratic function we can find the minimum in  $n$  steps, where  $n$  is the number of dimensions, by searching along conjugate directions. Conjugate directions are “orthogonal” with respect to the Hessian.

previous one. When the function steepens, the damping becomes larger, and vice versa.

---

**Example 4.17:** Conjugate gradient applied to the bean function

Minimizing the same bean function from Ex. 4.15 and the same line search algorithm and settings, we get the optimization path shown in Fig. 4.32. The changes in direction for the conjugate-gradient method are smaller than for steepest descent and it takes less iterations to achieve the same convergence tolerance.

---

#### 4.4.3 Newton's Method

The steepest descent and conjugate gradient methods use only first-order information (the gradient). Newton's method uses second-order information to enable better estimates of favorable search directions. The method is based on a Taylor's series expansion about the current design point:

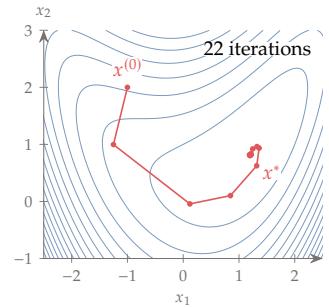
$$f(x^{(k)} + s^{(k)}) = f^{(k)} + \nabla f^{(k)} T s^{(k)} + \frac{1}{2} s^{(k)} T H^{(k)} s^{(k)} + \dots, \quad (4.51)$$

where  $s^{(k)}$  is some vector centered at  $x^{(k)}$ . We can find the step  $s^{(k)}$  that minimizes this quadratic model (ignoring the higher-order terms). We do this by taking the derivative with respect to  $s^{(k)}$  and setting that equal to zero:

$$\begin{aligned} \frac{df(x^{(k)} + s^{(k)})}{ds^{(k)}} &= \nabla f^{(k)} + H^{(k)} s^{(k)} = 0 \\ H^{(k)} s^{(k)} &= -\nabla f^{(k)} \\ s^{(k)} &= -H^{(k)}^{-1} \nabla f^{(k)}. \end{aligned} \quad (4.52)$$

Mathematically, we use the notation  $H^{-1}$ , but in a computational implementation one would typically not explicitly invert the matrix for efficiency reasons. Instead, one would solve the linear system  $H^{(k)} s^{(k)} = -\nabla f^{(k)}$ .

Using the same quadratic example from the previous sections, we see that Newton's method converges in one step (Fig. 4.33). This is not surprising. Because our function is quadratic, the quadratic "approximation" from the Taylor's series is exact, and so we can find the minimum in one step. For a general nonlinear function, it will take more iterations, but using curvature information should help us obtain a better estimate for a search direction compared to first-order



**Figure 4.32:** Conjugate gradient optimization path.

methods. Not only does Newton's method provide a better search direction, but it also provides a step length embedded in  $s^{(k)}$ , because the quadratic model provides an estimate of the stationary point location. Furthermore, Newton's method exhibits quadratic convergence.

While Newton's method is promising, in practice there are a few issues. Fortunately, we can address each of these challenges.

1. Problem: The Hessian might not be positive definite, in which case the search direction is not a descent direction.

Solution: Because we are not yet at the minimum, we know a descent direction exists. It is possible to modify the Hessian such that it is positive definite, and still prove convergence. The methods of the next section force positive definiteness by construction.

2. Problem: The predicted new point  $x^{(k)} + s^{(k)}$  is based on a second-order approximation and so may not actually yield a good point. In fact, the new point could be worse:  $f(x^{(k)} + s^{(k)}) > f(x^{(k)})$ .

Because the search direction  $s^{(k)}$  is a descent direction, if we back-track enough our search direction will yield a function decrease.

3. Problem: The Hessian can be difficult or costly to obtain.

Solution: This is unavoidable for Newton's method, but an alternative exists. (The quasi-Newton methods we discuss next).

---

#### Example 4.18: Newton method applied to the bean function

Minimizing the same bean function from Ex. 4.15 and Ex. 4.17, we get the optimization path shown in Fig. 4.34. Newton's method takes fewer iterations to achieve the same convergence tolerance.

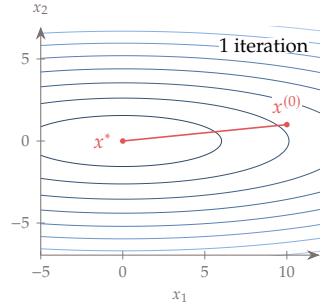
---

#### 4.4.4 Quasi-Newton Methods

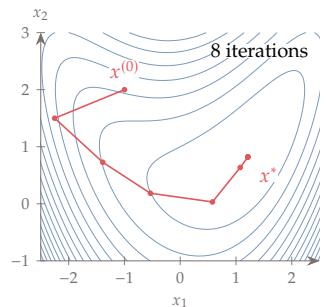
As discussed above, Newton's method is effective because the second-order information allows for better search directions, but it has the major shortcoming of requiring the Hessian in the first place. Quasi-Newton methods are designed to address this issue. The basic idea is that we can use first-order information (gradients) along each step in the iteration path to build an approximation of the Hessian.

In one dimension, we can use a secant line to estimate the slope of a curve (its derivative), which can be written as the finite difference

$$f' \approx \frac{f^{(k+1)} - f^{(k)}}{x^{(k+1)} - x^{(k)}}, \quad (4.53)$$



**Figure 4.33:** Iteration history for quadratic function using an exact line search and Newton's method. Unsurprisingly, only one iteration is required.



**Figure 4.34:** Newton optimization path.

where  $f'$  might be used to estimate the slope at  $k$  or  $k + 1$  (Fig. 4.35). We can use a similar finite difference to estimate the curvature using the slopes and rearrange the equation to get

$$f''^{(k+1)}(x^{(k+1)} - x^{(k)}) = f'^{(k+1)} - f'^{(k)}. \quad (4.54)$$

We use the same concept in  $n$  dimensions, but now the difference between the gradients at two different points yields a vector that represents an approximation of the curvature in that direction. Denoting the approximate Hessian as  $B$  and the step determined by the latest line search, which is  $s^{(k)} = x^{(k+1)} - x^{(k)} = \alpha^{(k)} p^{(k)}$ , we can write the *secant condition*

$$B^{(k+1)} s^{(k)} = \nabla f^{(k+1)} - \nabla f^{(k)}. \quad (4.55)$$

This states that the projection of the approximate Hessian onto  $s^{(k)}$  must yield the same curvature predicted by taking the difference between the gradients. The secant condition provides a requirement consisting of  $n$  equations where the step and the gradients are known. However, there are  $n(n + 1)/2$  unknowns in the approximate Hessian (recall that it is a symmetric matrix), so this is not sufficient to determine  $B$ . There is another requirement, which is that  $B$  must be positive definite. This yields another  $n$  but that still leaves us with an infinite number of possibilities for  $B$ .

Given that  $B$  must be positive definite, the secant condition (4.55) is only possible if the predicted curvature is positive along the step, that is,

$$s^{(k)}^T (\nabla f^{(k+1)} - \nabla f^{(k)}) > 0. \quad (4.56)$$

This is called the *curvature condition* which is automatically satisfied if the line search finds a step that satisfies the strong Wolfe conditions.

Davidon, Fletcher, and Powell devised an effective strategy to estimate the Hessian.<sup>13,14</sup> Because there is an infinite number of solutions, they formulated a way to select  $H$  by picking the one that was “closest” to the Hessian of the previous iteration, while still satisfying the requirements of the secant rule: symmetry and positive definiteness. This turns out to be an optimization problem in itself, but with an analytic solution. This led to the DFP method, which was a very impactful idea in the field of nonlinear optimization.

This method was soon superseded by the BFGS method developed by Broyden, Fletcher, Goldfarb, and Shannon<sup>57–60</sup> and so we focus on that method instead. They started with the observation that what we ultimately want is the inverse of the Hessian so that we can predict the next search direction:

$$p^{(k)} = -B^{(k)-1} \nabla f^{(k)}. \quad (4.57)$$

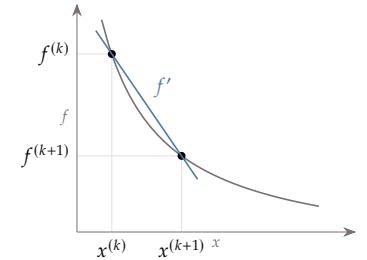


Figure 4.35: A secant line at point  $x^{(k)}$  used to estimate  $f'^{(k)}$ .

13. Davidon, *Variable Metric Method for Minimization*. 1991

14. Fletcher et al., *A Rapidly Convergent Descent Method for Minimization*. 1963

57. Broyden, *The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations*. 1970

58. Fletcher, *A new approach to variable metric algorithms*. 1970

59. Goldfarb, *A family of variable-metric methods derived by variational means*. 1970

60. Shanno, *Conditioning of quasi-Newton methods for function minimization*. 1970

Their key insight was that rather than estimating the Hessian, then solving a linear system, we should directly estimate the Hessian inverse instead. We will denote the Hessian inverse as  $V$  ( $V^{(k)} = H^{(k)^{-1}}$ ). Using the Hessian inverses changes our search prediction step to:

$$p^{(k)} = -V^{(k)}\nabla f^{(k)}. \quad (4.58)$$

Notice that once we have  $V^{(k)}$ , we only need to perform a matrix-vector multiplication, which is a much faster operation than solving a linear system.

The rest of the approach is similar to the DFP method, in the sense that we still need  $V$  to be symmetric, positive definite, satisfy the secant rule, and of all possible matrices, we will choose the one closest to the one from the previous iteration. The secant rule (4.55) can be rewritten in terms of our new estimate  $V^{(k+1)}$  as:

$$V^{(k+1)}(\nabla f^{(k+1)} - \nabla f^{(k)}) = x^{(k+1)} - x^{(k)}. \quad (4.59)$$

Mathematically, the problem that we need to solve to estimate the next Hessian inverse  $V^{(k+1)}$  is:

$$\begin{aligned} & \text{minimize} && \|V^{(k+1)} - V^{(k)}\| \\ & \text{by varying} && V^{(k+1)} \\ & \text{subject to} && V^{(k+1)} = V^{(k+1)^T} \\ & && V^{(k+1)} \left( \nabla f^{(k+1)} - \nabla f^{(k)} \right) = x^{(k+1)} - x^{(k)} \end{aligned} \quad (4.60)$$

Fortunately, this optimization problem can be solved analytically, depending on the choice of the matrix norm. The matrix norm used in the BFGS method is a weighted Frobenius norm, with a particular weighting (same one used in the DFP method). For further details see Fletcher<sup>61</sup>. The solution for  $V^{(k+1)}$  is:

$$V^{(k+1)} = \left[ I - \frac{s^{(k)}y^{(k)^T}}{s^{(k)^T}y^{(k)}} \right] V^{(k)} \left[ I - \frac{y^{(k)}s^{(k)^T}}{s^{(k)^T}y^{(k)}} \right] + \frac{s^{(k)}s^{(k)^T}}{s^{(k)^T}y^{(k)}}. \quad (4.61)$$

where

$$s^{(k)} = x^{(k+1)} - x^{(k)} = \alpha^{(k)} p^{(k)}$$

61. Fletcher, *Practical Methods of Optimization*. 1987

is the step that resulted from the last line search. The other important term is the estimate of the curvature in the direction of that line search, which is given by the difference between the gradients at the end and start of the line search (the last two major iterations),

$$y^{(k)} = \nabla f^{(k+1)} - \nabla f^{(k)}.$$

While the denominator  $s^T y^{(k)}$  is a dot product resulting in a scalar, the numerators  $s^{(k)} y^{(k)T}$  and  $y^{(k)} s^{(k)T}$  are an outer products that result in  $n_x \times n_x$  matrices of rank 1. The division of this matrix by the scalar is performed element wise.

Eq. 4.61 provides an analytic expression to update the inverse of the Hessian at each iteration. The advantages of this approximation are that we only need first-order information and that we do not need to evaluate points other than the iterations we are already performing. For the first iteration, we usually set  $V_0$  to the identity matrix, or a scaled version of it. Using the identity matrix for  $V_0$  in Eq. 4.58 results in

$$p_0 = -\nabla f_0 \quad (4.62)$$

and thus the first step is a steepest descent step. Subsequent iterations use information from the previous Hessian inverse, the direction and length of the last step, and the difference in the last two gradients to improve the estimate of the Hessian inverse.

The optimization problem (4.60) does not explicitly include a constraint on positive definiteness. It turns out that this update formula will always produce a  $V^{(k+1)}$  that is positive definite as long as  $V^{(k)}$  is positive definite. Therefore if we start with an identity matrix as suggested above, all subsequent updated produce positive definite matrices.

---

**Example 4.19:** BFGS applied to the bean function

---

Minimizing the same bean function from previous examples using BFGS, we get the optimization path shown in Fig. 4.36. We initialize the inverse Hessian to the identity matrix. Using the BFGS update procedure, after two iterations, with  $x^{(2)} = (0.065647, -0.219401)$ , the inverse Hessian approximation is

$$V^{(2)} = \begin{bmatrix} 0.320199 & -0.100560 \\ -0.100560 & 0.219681 \end{bmatrix}.$$

Compare this to the exact inverse Hessian at the same point,

$$H^{-1}(x^{(2)}) = \begin{bmatrix} 0.345784 & 0.015133 \\ 0.015133 & 0.167328 \end{bmatrix}$$

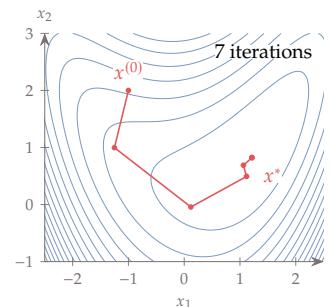
The predicted curvature is reasonable, but not accurate.

By the end of the optimization, at  $x^* = (1.213336, 0.824181)$ , the BFGS estimate is

$$V^* = \begin{bmatrix} 0.276503 & 0.224956 \\ 0.224956 & 0.346879 \end{bmatrix},$$

while the exact one is

$$H^{-1}(x^*) = \begin{bmatrix} 0.276965 & 0.224034 \\ 0.224034 & 0.347886 \end{bmatrix}.$$



**Figure 4.36:** BFGS optimization path.

Now the estimate is much more accurate.

---

#### 4.4.5 Limited Memory Quasi-Newton Methods

As the number of design variables becomes large, then even just storing the Hessian inverse matrix in memory can require an excessive amount of resources. For example, if there are millions or billions of design variables the memory requirements can be prohibitive, but even for problems with 100 design variables the techniques of this section are often used to improve computational efficiency with minimal sacrifice in accuracy. Recall that we are only interested in the matrix vector product  $V\nabla f$ . We can compute that product without ever actually forming the matrix  $V$ . We will discuss how this is done with the BFGS update as this is the most popular approach (known as L-BFGS), although other Quasi Newton update formulas can also be used.

Notice that Eq. 4.61 defines a recursive sequence. As shorthand we define the scalar:

$$\sigma = \frac{1}{s^T y} \quad (4.63)$$

then the BFGS update is given by:

$$V^{(k)} = [(I - \sigma s y^T) V (I - \sigma y s^T) + \sigma s s^T]^{(k-1)} \quad (4.64)$$

If we saved the sequence of  $s$  and  $y$  vectors, and specified a starting value for  $V^{(0)}$ , then we could compute any subsequent  $V^{(k)}$ . Of course, what we really want is  $V^{(k)} \nabla f^{(k)}$ . This product can be computed algorithmically using the recurrence relationship.

However, as described, the new algorithm doesn't provide much benefit yet. The goal was to avoid storing a large dense matrix, and instead we are storing a long sequence of vectors and a starting matrix. To alleviate this issue, we don't store the entire history, but rather only store the last  $m$  vectors for  $s$  and  $y$ . In practice,  $m$  is usually between 5–20. Next, we make the starting Hessian diagonal so it only requires vector storage, or scalar storage if we make all entries in the diagonal equal. A common choice is to use a scaled identity matrix, which just requires storing one number:

$$V^{(0)} = \frac{s^T y}{y^T y} I \quad (4.65)$$

where the  $s$  and  $y$  values on the right hand side would use the previous iteration. The algorithm is summarized in Alg. 4.20.

---

**Algorithm 4.20:** Compute the product of inverse Hessian and a vector using the BFGS update rule

**Inputs:**

$\nabla f^{(k)}$ : Gradient at point  $x^{(k)}$   
 $s^{(k-1, \dots, k-m)}$ : History of steps  $x^{(k)} - x^{(k-1)}$   
 $y^{(k-1, \dots, k-m)}$ : History of gradient differences  $\nabla f^{(k)} - \nabla f^{(k-1)}$

**Outputs:**

$d$ : Desired product  $V^{(k)} \nabla f^{(k)}$

---

```

 $d = \nabla f^{(k)}$ 
for  $i = k - 1$  to  $k - m$  by  $-1$  do
     $\alpha^{(i)} = \sigma^{(i)} s^{(i)T} d$ 
     $d = d - \alpha^{(i)} y^{(i)}$ 
end for
 $V^{(0)} = \begin{pmatrix} s_{k-1}^T y_{k-1} \\ y_{k-1}^T y_{k-1} \end{pmatrix} I$ 
 $d = V^{(0)} d$ 
for  $i = k - m$  to  $k - 1$  do
     $\beta^{(i)} = \sigma^{(i)} y^{(i)T} d$ 
     $d = d + (\alpha^{(i)} - \beta^{(i)}) s^{(i)}$ 
end for

```

---

We no longer need to bear the memory cost of storing a large matrix, or the computational cost of a large matrix-vector product. Instead, we store only a small number of vectors, and require only a small number of vector-vector products (a cost that scales linearly with  $n$  rather than quadratically).

---

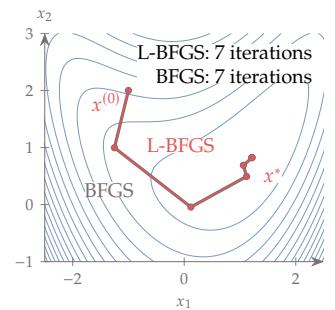
**Example 4.21:** L-BFGS compared to BFGS for the bean function

Minimizing the same bean function from the previous examples, the optimization iterations using BFGS and L-BFGS are the same, and are shown in Fig. 4.37. The L-BFGS method is applied to the same sequence using the last 5 iterations. The number of variables is too small to benefit from the limited memory approach, but we show it in this small problem as an example. At the same  $x^*$  as in Ex. 4.19, the product  $VVf$  is estimated using Alg. 4.20 as:

$$d^* = \begin{bmatrix} -7.38683 \times 10^{-5} \\ 5.75370 \times 10^{-5} \end{bmatrix}$$

whereas the exact value is:

$$V^* \nabla f^* = \begin{bmatrix} -7.49228 \times 10^{-5} \\ 5.90441 \times 10^{-5} \end{bmatrix}$$



**Figure 4.37:** Optimization paths using BFGS and L-BGFS.

---

**Example 4.22:** Total potential energy contours of spring system

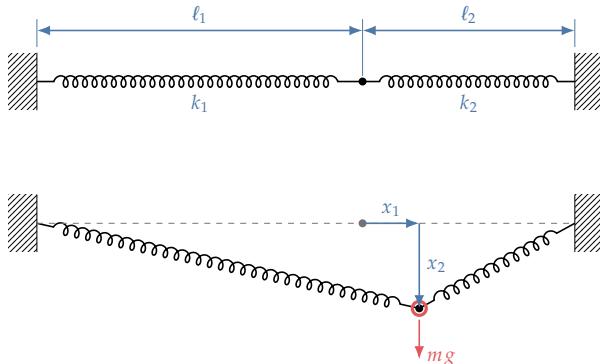
Many structural mechanics models involve solving an unconstrained energy minimization problem. Consider a mass supported by two springs, as shown in Fig. 4.38. Formulating the total potential energy for the system as a function of the mass position yields the problem,

$$\text{minimize} \quad \frac{1}{2}k_1 \left( \sqrt{(\ell_1 + x_1)^2 + x_2^2} - \ell_1 \right)^2 + \frac{1}{2}k_2 \left( \sqrt{(\ell_2 - x_1)^2 + x_2^2} - \ell_2 \right)^2 - mgx_2$$

by varying  $x_1, x_2$

(4.66)

The contours of this function are shown in Fig. 4.39 for the case where



**Figure 4.38:** Two-spring system with no applied force (top) and with applied force (bottom).

$\ell_1 = 12, \ell_2 = 8, k_1 = 1, k_2 = 10, mg = 7$ . There is both a minimum and a maximum. The minimum represents the position of the mass at the stable equilibrium condition. The maximum also represents an equilibrium point, but it is unstable. Starting near the maximum, steepest descent, conjugate gradient, and quasi-Newton all converge to the minimum. As expected, steepest descent is the least efficient and quasi-Newton is the most efficient.

---

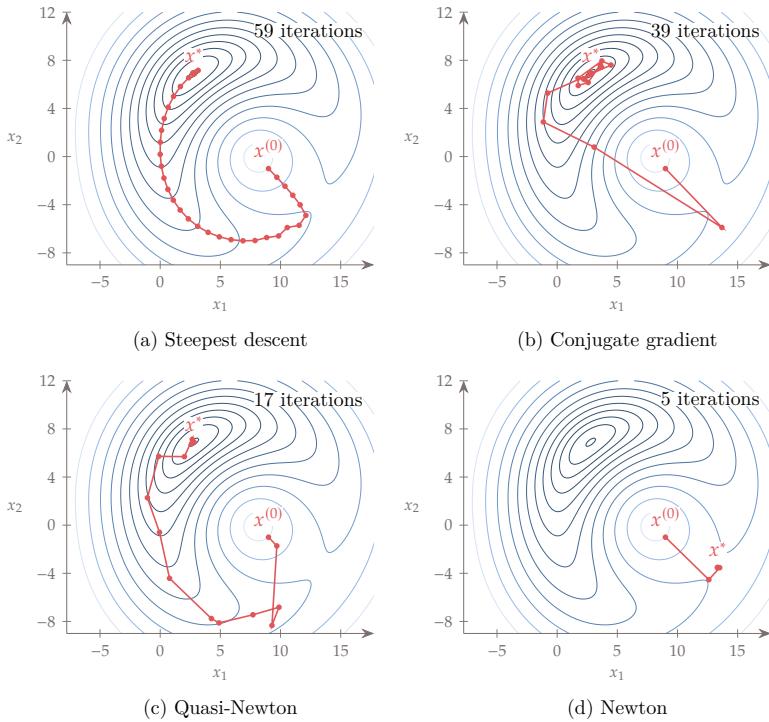
**Example 4.23:** Comparing methods for the Rosenbrock function

We now test the methods on the more challenging function,

$$f(x_1, x_2) = (1 - x_1)^2 + 100 \left( x_2 - x_1^2 \right)^2,$$

which is known as the Rosenbrock function. This is a well-known optimization problem because a narrow highly curved valley makes it challenging to minimize. § The convergence history for four methods starting from  $x = (-1.2, 1.0)$

§The “bean” function we used in previous examples is a milder version of the Rosenbrock function.



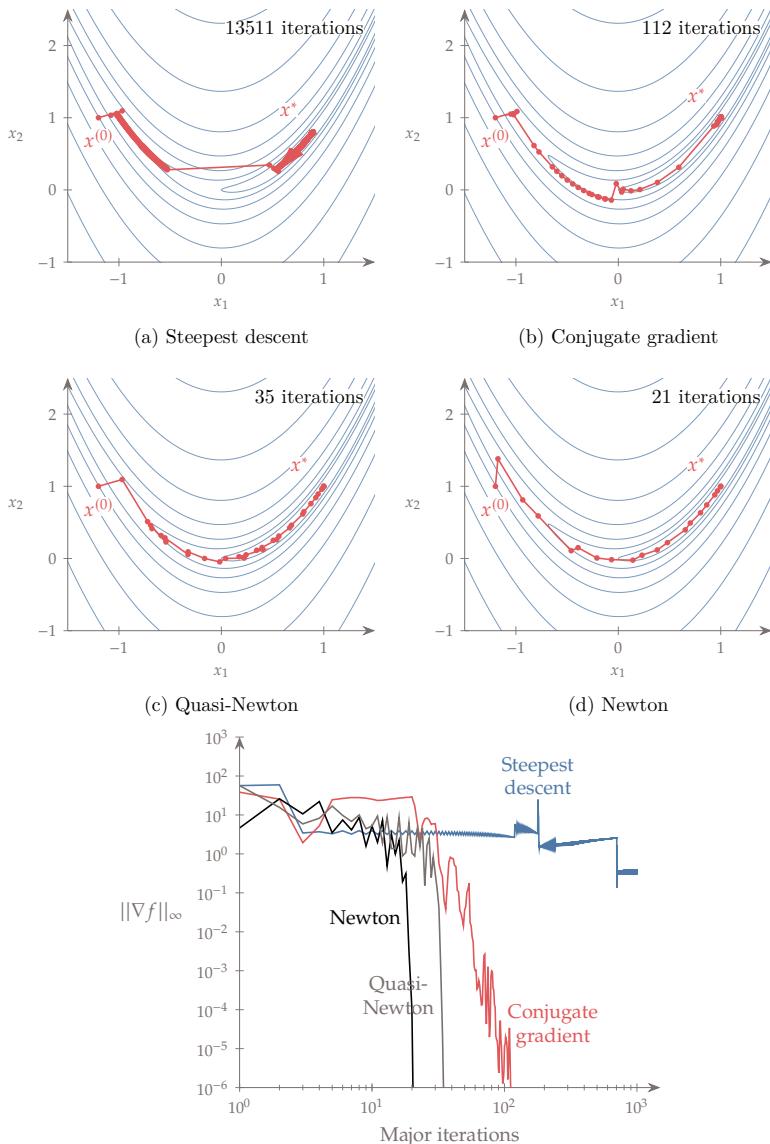
**Figure 4.39:** Minimizing the total potential for two-spring system.

is shown in Fig. 4.41. All four methods use an inexact line search using the same parameters and a convergence tolerance of  $\|\nabla f\|_\infty \leq 10^{-6}$ . Compared to the previous two examples, the difference between the steepest descent and the other methods is much more dramatic (two orders of magnitude more iterations!), owing to the more challenging variation in the curvature (recall Ex. 4.14).

Steepest descent does converge takes a large number of iterations because it bounces between the steep walls of the valley. One of the line search gets lucky and takes a shortcut to another part of the valley, but even then it cannot make up for its inherent inefficiency. The conjugate gradient method is much more efficient because it damps the steepest descent oscillations with a contribution from the previous direction. Eventually, conjugate gradient achieves superlinear convergence near the optimum, which saves many iterations to get the last several orders of magnitude in the convergence criterion. The methods that use second-order information are even more efficient, exhibiting quadratic convergence in the last few iterations.

---

The number of major iterations is not always an effective way to compare performance. For example, Newton's method takes fewer major iterations, but each iteration in Newton's method is more expensive than each iteration in the quasi-Newton method. This is because New-



**Figure 4.40:** Optimization paths for the Rosenbrock function using steepest descent, conjugate gradient, Newton, and BFGS

**Figure 4.41:** Convergence of the four methods shows the dramatic difference between the linear convergence of steepest descent, the superlinear convergence of the conjugate gradient method, and the quadratic convergence of the methods that use second-order information.

ton's method requires a linear solution, which is an  $O(n^3)$  operation, as opposed to a matrix-vector multiplication, which is an  $O(n^2)$  operation. For a small problem like the Rosenbrock function this is an insignificant difference, but for large problems this is a significant difference in time. Additionally, each major iteration includes a line search, and depending on the quality of the search direction, the number of function calls contained in each iteration will differ.

**Tip 4.24:** Gradient-based optimization can find the global optimum.

Gradient-based methods are local search methods. If the design space is fundamentally multimodal, it may be useful to augment the gradient-based search with a global search.

The simplest and most common approach is to use a *multistart* approach, where we run a gradient-based search multiple times, starting from different points. The starting points might be chosen from engineering intuition, randomly generated points, or sampling methods, such as Latin hypercube sampling (see Chapter 10).

Convergence testing is needed to determine a suitable number of starting points. If all points converge to the same optimum, and the starting points were well spaced, this suggest that the design space might not be multimodal after all. By using multiple starting points, we increase the likelihood that we find the global optimum, or at least that we find a better optimum than would be found with a single starting point. One advantage of this approach is that it can easily be run in parallel.

Another approach is to start with a global search strategy, like a population-based gradient-free algorithm (see Chapter 7). After some suitable initial exploration, the designs in the population become starting points for gradient-based optimization. This approach allows us to find points that satisfy the optimality conditions, which is typically not possible with a pure gradient-free approach. It also improves the convergence rate and finds the optima more precisely.

---

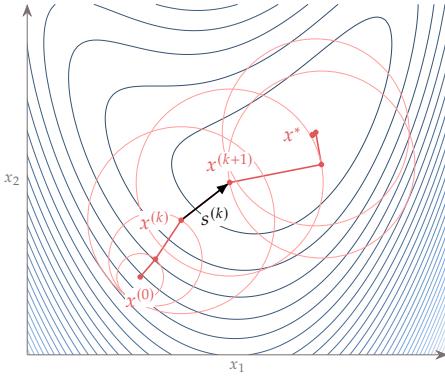
## 4.5 Trust-region Methods

Trust-region methods, also known as restricted step methods, present an alternative to the line-search based algorithms presented so far. The motivation for trust-region methods is to address the issues caused by non-positive definite Hessian matrices in Newton and quasi-Newton methods, as well as other weaknesses.

Unconstrained optimization algorithms based on a line search consist of determining a search direction, then solving the line search subproblem, which determines the distance to move along that direction. Trust-region methods are fundamentally different. Instead of fixing the direction and then finding the distance, trust-region methods fix the maximum distance, and then find the direction and distance that yield the most improvement. The trust-region method requires a model of the function to be minimized, and the definition of a region within which we *trust* the model to be good enough for our purposes. The most common model is a local quadratic function, but other models may also be used. The trust-region is centered about the current iteration point, and can be defined as an  $n$ -dimensional box, sphere, or ellipsoid of a

given size. Each trust-region iteration consists in the following main stages:

1. Update the function model (e.g., quadratic)
2. Minimize the model within the trust region
3. Update the trust-region size and location



**Figure 4.42:** Trust region approach for globalization. In this case, where the trust regions are circular in this case.

The trust-region subproblem is

$$\begin{aligned} & \text{minimize } \tilde{f}(s) \\ & \text{by varying } s \\ & \text{subject to } \|s\| \leq \Delta, \end{aligned} \tag{4.67}$$

where  $\tilde{f}(s)$  is the local trust-region model,  $s$  is the step from the current iteration point, and  $\Delta$  is the size of the trust region. Note that we use the notation  $s$  instead of  $p$  to indicate that this is a step vector (direction and magnitude) and not just the direction  $p$  used in line search based methods.

The subproblem above defines the trust-region as a norm. The Euclidean norm,  $\|s\|_2$ , defines a spherical trust region and is the most common type of trust region. Sometimes  $\infty$ -norms are used instead as they are easy to apply, but 1-norms are rarely used as they are just as complex as 2-norms but introduce sharp corners that are sometimes problematic<sup>62</sup>. The shape of the trust region, dictated by the norm, can have a significant impact on the convergence rate. The ideal trust region shape depends on the local function space and some algorithms allow for the trust region shape to change throughout the optimization.

62. Conn et al., *Trust Region Methods*. 2000

Using a quadratic trust-region model and the Euclidean norm we can define the more specific subproblem

$$\begin{aligned} \text{minimize } & \tilde{f}(s) = f^{(k)} + \nabla f^{(k)^T} s + \frac{1}{2} s^T B^{(k)} s \\ \text{subject to } & \|s\|_2 \leq \Delta^{(k)}, \end{aligned} \quad (4.68)$$

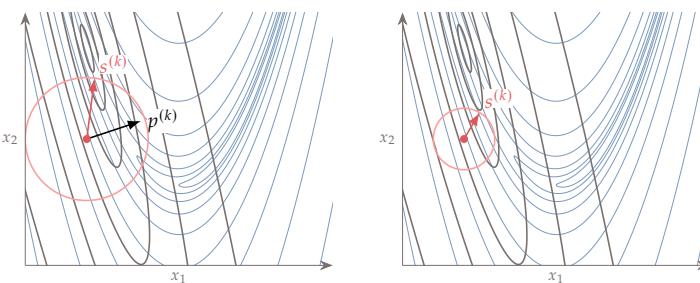
where  $B^{(k)}$  is the approximate Hessian at our current iterate. This problem has a quadratic objective and quadratic constraints and is called a quadratically constrained quadratic program (QCQP). If the problem is unconstrained and  $B$  is positive definite, we can get to the solution using a single step  $s = -B^{(k)^{-1}} \nabla f^{(k)}$ . However, due to the constraints, there is no analytic solution for the QCQP. While the problem is still straightforward to solve numerically (it is a convex problem, see Chapter 11), it requires an iterative solution approach with multiple factorizations. Similarly to the line search, where we only obtain a sufficiently good point instead of finding the exact minimum, in the trust-region subproblem we seek an approximate solution to the QCQP. The inclusion of the trust-region constraint allows us to omit the requirement that  $B^{(k)}$  be positive definite, which is used in most of the quasi-Newton methods. We do not detail approximate solution approaches to the QCQP but multiple algorithms exist.<sup>56,62,63</sup>

The left side of Fig. 4.43 shows an example of function contours for the Rosenbrock function, a local quadratic model (in blue), and a spherical trust-region (red circle). The trust-region step seeks the minimum of the local quadratic model within the spherical trust region. Notice on the right side that, unlike line search methods, as the size of the trust region changes the direction of the step also change (the solution to Eq. 4.68).

56. Nocedal et al., *Numerical Optimization*. 2006

62. Conn et al., *Trust Region Methods*. 2000

63. Steihaug, *The Conjugate Gradient Method and Trust Regions in Large Scale Optimization*. 1983



**Figure 4.43:** The blue contour lines are the Rosenbrock function and the gray contours are a local quadratic approximation about the current iteration (where the arrow originate). The red circle represents a trust region. It is a safeguard to prevent steps beyond where the local model is likely to be valid. The trust-region step  $s^{(k)}$  finds the minimum of the blue contours while remaining within the trust-region boundary.

### 4.5.1 Trust Region Sizing Strategy

This section presents an algorithm for updating the size of the trust region at each iteration. The trust region can grow, shrink, or remain the same, depending on how well the model predicts the actual function decrease. The metric we use to assess the model is the actual function decrease divided by the expected decrease

$$r = \frac{f(x) - f(x + s)}{\tilde{f}(0) - \tilde{f}(s)}. \quad (4.69)$$

The denominator in this definition is the expected decrease, which is always positive. The numerator is the actual change in the function, which could be a reduction or an increase. An  $r$  value close to unity means that the model agrees well with the actual function. An  $r$  value larger than one is fortuitous and means the actual decrease was even greater than expected. A negative value of  $r$  means that the function actually increased at the expected minimum, and therefore the model is not suitable.

The trust region sizing strategy detailed in Alg. 4.25 determines the size of the trust region at each major iteration  $k$  based on the value of  $r^{(k)}$ . The parameters in this algorithm are not derived from any theory but are rather empirical. This example uses the basic procedure from Nocedal *et al.*<sup>56</sup>, but with recommended parameters from Conn *et al.*<sup>62</sup>. The initial value of  $\Delta$  is usually 1 assuming the problem is already well scaled. One way to rationalize the trust-region method is that the quadratic approximation of a nonlinear function is in general reasonable only within a limited region around the current point  $x^{(k)}$ . We can overcome this limitation by minimizing the quadratic function within a region around  $x^{(k)}$  within which we *trust* the quadratic model. When our model performs well, we expand the trust region. When it performs poorly we shrink the trust region. If we shrink the trust region sufficiently, our local model should eventually be a good approximation of the real function, as dictated by the Taylor series expansion. We should also set a maximum trust region size,  $\Delta_{\max}$  to prevent the trust region from expanding too much. Otherwise, if we have good fits over part of the design space, it may take too long to reduce the trust region size to an acceptable size over other portions of the design space where a smaller trust region is needed. The same stopping criteria used in other gradient-based methods are applicable.<sup>¶</sup>

<sup>56</sup>. Nocedal *et al.*, *Numerical Optimization*. 2006

<sup>62</sup>. Conn *et al.*, *Trust Region Methods*. 2000

---

Algorithm 4.25: Trust-region algorithm

Inputs:

<sup>¶</sup>Conn provides more detail on trust-region problems including trust region norms and scaling, approaches to solving the trust-region subproblem, extensions to the model, and other important practical considerations.

$x^{(0)}$ : Starting point

$\Delta^{(0)}$ : Initial size of the trust region

### Outputs:

$x^*$ : Optimal point

**while** not converged **do**

    Compute or estimate the Hessian

    Solve (approximately) for  $s^{(k)}$

        Using (4.67)

    Compute  $r^{(k)}$

        Using (4.69)

*> Resize trust region*

**if**  $r^{(k)} \leq 0.05$  **then**

*Poor model*

$\Delta^{(k+1)} = \Delta^{(k)} / 4$

*Shrink trust region*

$s^{(k)} = 0$

*Reject step*

**else if**  $r^{(k)} \geq 0.9$  and  $\|s^{(k)}\| = \Delta^{(k)}$  **then**

*Good model and step to edge*

$\Delta^{(k+1)} = \min(2\Delta^{(k)}, \Delta_{\max})$

*Expand trust region*

**else**

*Reasonable model and step within trust region*

$\Delta^{(k+1)} = \Delta^{(k)}$

*Maintain trust region size*

**end if**

$x^{(k+1)} = x^{(k)} + s^{(k)}$

*Update location of trust region*

$k = k + 1$

*Update iteration count*

**end while**

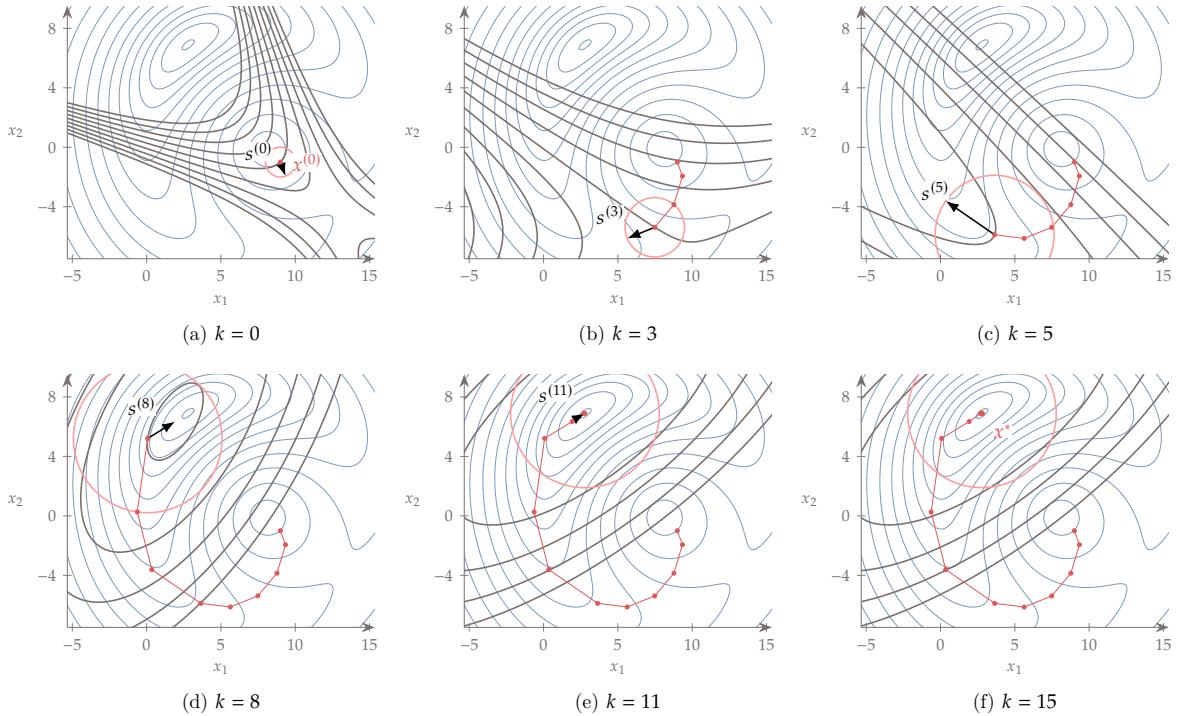
---

**Example 4.26:** Minimizing total potential energy of spring system with trust region

Minimizing the total potential energy function from Ex. 4.22 using the a trust-region method starting from the same points as before yields the optimization path shown in Fig. 4.44. The initial trust region size is  $\Delta = 0.3$  and the maximum allowable is  $\Delta = 1.5$ . The convergence criteria is based on the difference in subsequent iterations, such that  $\|x^{(k)} - x^{(k-1)}\| \leq 10^{-8}$ . The first few quadratic approximations do not have a minimum because the function has negative curvature around the starting point, but the trust region prevents steps that are too large. When it gets close enough to the bowl containing the minimum, the quadratic approximation has a minimum and the trust region subproblem yields a minimum within the trust region. In the last few iterations, the quadratic is a good model and therefore the region remains large.

### 4.5.2 Comparison with Line Search Methods

Generally speaking, trust-region methods are more strongly dependent on accurate Hessians than line search methods. For this reason, they are usually only effective when exact gradients (or better yet an exact



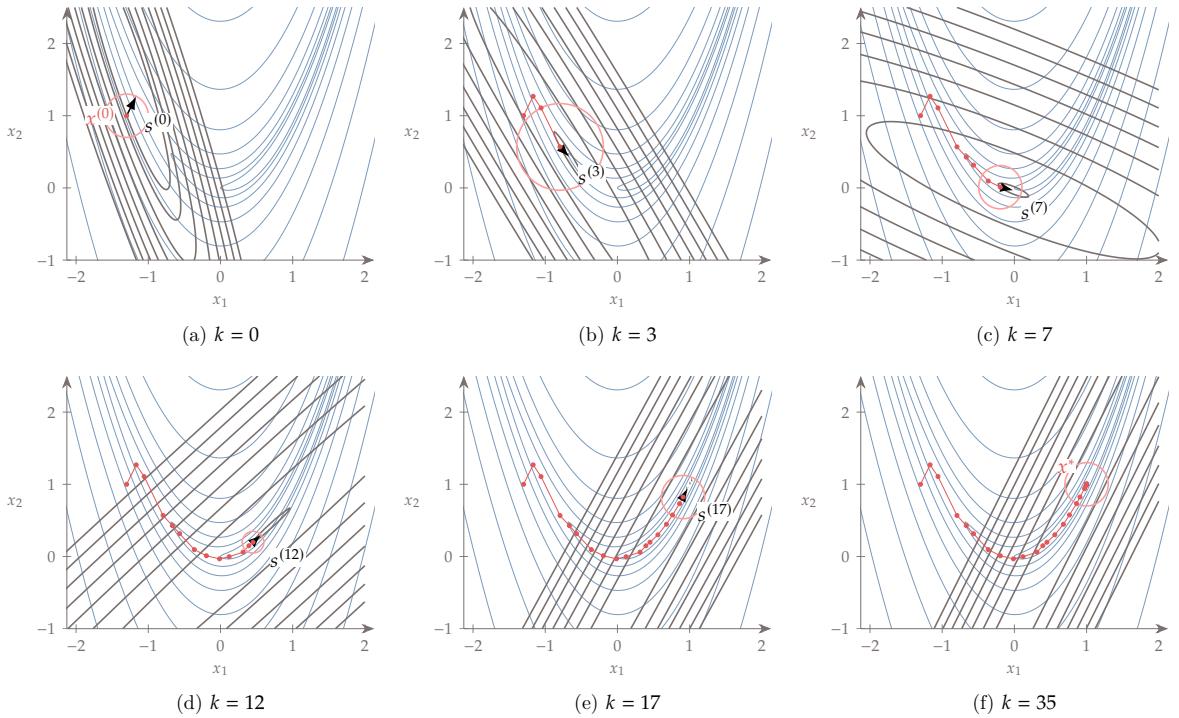
**Figure 4.44:** Minimizing the total potential for two-spring system using a trust-region method.

Hessian) can be supplied. In fact several optimization packages require the user to provide the full Hessian in order to use a trust-region approach. Trust-region methods generally require fewer iterations than quasi-Newton methods but each iteration is more computationally expensive because of the need for at least one matrix factorization.

Scaling can also be more challenging with trust-region approaches. Newton's method is invariant with scaling, but the use of a Euclidean trust-region constraint implicitly assumes that the function changes in each direction at a similar rate. Some enhancements try to address this issue through the use of elliptical trust regions rather than spherical ones.

#### Example 4.27: Minimizing the Rosenbrock function using trust region

We now test the trust-region method on the Rosenbrock function. The overall path is similar to the other second-order methods. The initial trust region size is  $\Delta = 1$  and the maximum allowable is  $\Delta = 5$ . The convergence criteria is based on the difference in subsequent iterations, such that  $\|x^{(k)} - x^{(k-1)}\| \leq 10^{-8}$ . At any given point, the direction of maximum curvature of the quadratic approximation matches the maximum curvature across the valley and rotates as we track the bottom of the valley toward the minimum.



**Tip 4.28:** Accurate derivatives matter.

The effectiveness of gradient-based methods depends strongly on providing accurate gradients. Convergence difficulties, or apparent multimodal behavior, are often mistakenly identified as fundamental modeling issues when in reality the numerical issues are caused by inaccurate gradients. Chapter 6 is devoted to the subject of obtaining accurate derivatives.

**Figure 4.45:** Minimization of the Rosenbrock function using a trust region method.

## 4.6 Summary

Unsurprisingly, the gradient plays a crucial role in gradient-based optimization: it points in the direction of steepest increase, and when the gradient is zero we know we have reached a stationary point. If we use descent directions in our search then the stationary point will be a local minimum. Although the negative gradient points in the direction of steepest descent, we have seen that following this direction is generally not the best approach because it is prone to oscillation. Second-order methods (e.g., Newton's method) use curvature information to greatly improve the rate of convergence. Because supplying second derivatives

is often prohibitive, we typically use quasi-Newton methods where the Hessian is estimated from changes in the gradients.

All gradient-based methods are characterized by how they choose the search direction  $p^{(k)}$ . The four methods we have discussed yield the following search directions:

$$\begin{aligned} \text{Steepest descent: } & p^{(k)} = -\nabla f^{(k)} \\ \text{Conjugate gradient: } & p^{(k)} = -\nabla f^{(k)} + \beta^{(k)} p^{(k-1)} \\ \text{Newton: } & p^{(k)} = -H^{(k)^{-1}} \nabla f^{(k)} \\ \text{Quasi-Newton: } & p^{(k)} = -V^{(k)} \nabla f^{(k)} \end{aligned} \quad (4.70)$$

After determining an appropriate descent direction we need to determine how far to travel in that direction. This is the purpose of a line search. Although one might ideally want to find the minimum along that line, this turns out to be wasteful and instead we rely on criteria that define when the point is “good enough”. These criteria look for sufficient decrease, and a flattening of the curvature. Once a point satisfies these conditions we repeat the process and select a new search direction.

Trust regions provide an alternative approach where instead of using line searches, we define a region (typically a sphere), and solve a surrogate optimization problem within that region. Line search methods are more commonly used, but trust region approaches can be particularly effective if we are able to supply second derivatives.

## Problems

4.1 Answer *true* or *false* and justify your answer.

- a) Gradient-based optimization requires the function to be continuous and infinitely differentiable.
- b) Gradient-based methods perform a local search.
- c) Gradient-based methods are only effective for problems with one minimum.
- d) When the gradient is projected into a given direction, we get a vector pointing aligned with that direction.
- e) The Hessian of a unimodal function is positive-definite or positive-semidefinite everywhere.
- f) Each column  $j$  of the Hessian quantifies the rate of change of component  $j$  of the gradient vector with respect to all coordinate directions  $i$ .

- g) If the function curvature at a point is zero in some direction, that point cannot be a local minimum.
- h) A globalization strategy in a gradient-based algorithm ensures convergence to the global minimum.
- i) The goal of the line search is to find the minimum along a given direction.
- j) For minimization, the line search must always start in a descent direction.
- k) The direction in the steepest descent algorithm for a given iteration is orthogonal to the direction of the previous iteration.
- l) Newton's method is not affected by problem scaling.
- m) Quasi-Newton methods approximate the function Hessian by finite-differencing gradients.
- n) Overall, Newton's with a line search is the best choice among gradient-based methods because it uses exact second-order information.
- o) The trust region method does not require a line search.

4.2 Consider the function

$$f(x_1, x_2, x_3) = x_1^2 x_2 + 4x_2^4 - x_2 x_3 + x_3^{-1}.$$

Find the gradient of this function. Where is the gradient not defined? Calculate the directional derivative of the function at  $x_A = (2, -1, 5)$  in the direction  $p = [6, -2, 3]$ . Find the Hessian of this function. Is the curvature in the direction  $p$  positive or negative? Write the second-order Taylor's series expansion of this function. Plot the Taylor series values along the  $p$  direction and compare it to the actual function. Plot the contours of the Taylor series expansion about  $x_A$  and compare them to the contours of the original function. Zoom in your plot around  $x_A$  until the two sets of contours are indistinguishable. What is the order of magnitude in  $x$  did you end up with?

4.3 Consider the function from Ex. 4.1,

$$f(x_1, x_2) = x_1^3 + 2x_1 x_2^2 - x_2^3 - 20x_1. \quad (4.71)$$

Find the critical points of this function analytically and classify them. What is the global minimum of this function?

4.4 Review Kepler's wine barrel story from Section 2.2. Approximate the barrel as a cylinder and find the height and diameter of a barrel that maximizes its volume for a diagonal measurement of 1 m.

4.5 Consider the function:

$$f = x_1^4 + 3x_1^3 + 3x_2^2 + 6x_1x_2 + 6x_2 - 8x_2.$$

Find the critical points analytically and classify them. Where is the global minimum? Plot the function contours to verify your results.

4.6 Consider a slightly modified version of the function from Prob. 4.5, where we add a  $x_2^4$  term to get

$$f = x_1^4 + x_2^4 + 3x_1^3 + 3x_2^2 + 6x_1x_2 + 6x_2 - 8x_2.$$

Can you find the critical points analytically? Plot the function contours. Locate the critical points graphically and classify them.

4.7 *Line search algorithm implementation.* Implement the two line search algorithms from Section 4.3, such that they work in  $n$  dimensions ( $x$  and  $p$  can be vectors of any size).

- a) As a first test for your code, reproduce the results from the examples in Section 4.3 and plot the function and iterations for both algorithms. For the line search that satisfies the strong Wolfe conditions, reduce the value of  $\mu_2$  until you get an exact line search. How much accuracy can you achieve?
- b) Test your code on another easy two-dimensional function, such as the bean function from Ex. 4.15, starting from different points and using different directions (but remember that you must always provide valid descent direction, otherwise the algorithm might not work!). Does it always find a suitable point? Exploration: Try different values of  $\mu_2$  and  $\rho$  to analyze their effect on the number of iterations.
- c) Apply your line search algorithms to the two-dimensional Rosenbrock function and then the  $n$ -dimensional variant (see Appendix C.1.2). Again, try different points and search directions to see how robust the algorithm is, and try to tune  $\mu_2$  and  $\rho$ .

4.8 *Effect of scaling on line search.* Consider the one-dimensional function,

$$f(x) = -x + \gamma x^2,$$

where  $\gamma > 0$  is a parameter. This problem demonstrates the impact of poor scaling, which often an issue in practical optimization problems. Use your line search algorithm to investigate the effect of function curvature by using three values of  $\gamma$ : 0.5, 10, and  $10^4$ . Start from  $x = 0$  using a reduction parameter value of  $\rho = 0.5$ , a sufficient decrease parameter of  $\mu_1 = 10^{-6}$ , and an initial step length parameter of  $\alpha = 1$ . Your search direction in this case is just a scalar,  $p = 1$ .

- a) How many function evaluations are required for the three different values of  $\gamma$  for each of the algorithms? Observe and explain the trends.
- b) For the highest value of  $\gamma$ , scale  $x$  to make the problem better conditioned and verify that it works.
- c) Exploration: Try different starting points (you might have to set  $p = -1$ , depending on the direction of descent).

4.9 *Optimization algorithm implementation.* Program the steepest descent, conjugate gradient, and BFGS algorithms from Section 4.4. You must have a thoroughly tested line search algorithm from the previous exercise first. For the gradients, differentiate the functions analytically and compute them exactly. Solve each problem using your implementations of the various algorithms, as well as off-the-shelf optimization software for comparison.

- a) For your first test problem, reproduce the results from the examples in Section 4.4.
- b) Minimize the two-dimensional Rosenbrock function (see Appendix C.1.2) using the various algorithms and compare your results starting from  $x = (-1, 2)$ . Compare the total number of evaluations. Compare the number of minor versus major iterations. Discuss the trends. Exploration: Try different starting points and tuning parameters (e.g.,  $\rho$  and  $\mu_2$  in the line search) and compare the number of major and minor iterations.
- c) Benchmark your algorithms on the  $n$ -dimensional variant of the Rosenbrock function (see Appendix C.1.2). Try  $n = 3$  and  $n = 4$  first, then  $n = 8, 16, 32, \dots$ . What is the highest number of dimensions you can solve? How does the number of function evaluations scale with the number of variables?
- d) Optional: Implement L-BFGS and compare it with BFGS.

- 4.10 *Trust region implementation.* Implement a trust-region algorithm and apply it to one or more of the test problems from the previous exercise. Compare the trust region results with BFGS and the off-the-shelf software.
- 4.11 *Aircraft wing design.* We will solve the aircraft wing design problem described in Appendix C.1.6.
- 4.12 *Brachistochrone problem* The brachistochrone problem seeks to find the path that minimizes travel time between two points for a particle under the force of gravity (think of a bead constrained to slide on a wire whose shape you control). This was mentioned in Section 2.2 as one of the problems that inspired the developments in calculus of variations. Solve the discretized version of this problem (see Appendix C.1.7 for a detailed description).
- Plot the optimal path for the frictionless case with  $n = 10$  and compare it to the exact solution (see Appendix C.1.7).
  - Solve the optimal path with friction and plot the resulting path. Report the travel time between the two points and compare it to the frictionless case.
  - Study the effect of increased problem dimensionality. Start with 4 points and double the dimension each time up to 128 points. Plot and discuss the increase in computational expense with problem size. Example metrics include the number of major iterations, function evaluations, and computational time. Hint: When solving the higher-dimensional cases, start with the solution interpolated from a lower-dimensional case—this is called a *warm start*.

Engineering design optimization problems are rarely unconstrained. In this chapter, we explain how to solve constrained problems. The methods in this chapter build on the gradient-based unconstrained methods from Chapter 4 and also assume smooth functions. We first introduce the optimality conditions for a constrained optimization problem and then focus on three main types of methods for handling constraints: penalty functions, sequential quadratic optimization, and interior-point methods.

Penalty methods are no longer used in constrained gradient-based optimization because they have been replaced by more effective methods, but the concept of a penalty is useful when thinking about constraints, and is used as part of more sophisticated methods.

Sequential quadratic optimization and interior-point methods represent the state-of-the-art in nonlinear constrained optimization. We introduce the basics for these two methods, but a complete and robust implementation of these two methods requires rather detailed knowledge of a growing body of literature that is not covered here. There are many other methods not covered in this chapter, but they are either less effective, or are only more effective for certain problems.

At the end of the chapter, we discuss merit functions and filters. These considerations are an important part of the line search in both constrained optimization approaches.

By the end of this chapter you should be able to:

1. Describe the mathematical definition of optimality for a constrained problem.
2. Understand the motivation for and limitations of penalty methods.
3. Understand the concepts behind state-of-the-art constrained optimization algorithms and be able to use them to solve real engineering problems.

## 5.1 Constrained Problem Formulation

We can express a general constrained optimization problem as

$$\begin{aligned}
 & \text{minimize} && f(x) \\
 & \text{by varying} && x_i \quad i = 1, \dots, n_x \\
 & \text{subject to} && g_j(x) \leq 0 \quad j = 1, \dots, n_g \\
 & && h_k(x) = 0 \quad k = 1, \dots, n_h \\
 & && \underline{x}_i \leq x_i \leq \bar{x}_i \quad i = 1, \dots, n_x
 \end{aligned} \tag{5.1}$$

where the  $g_j(x)$  are the *inequality constraints*,  $h_k(x)$  are the *equality constraints*, and  $\underline{x}$  and  $\bar{x}$  are lower and upper *bound constraints* on the design variables. Both objective and constraint functions can be nonlinear, but they should be  $C^2$  continuous to be solved using gradient-based optimization algorithms. The inequality constraints are expressed as “less than” without loss of generality because they can always be converted to “greater than” by putting a negative sign on  $g_j$ . We could also eliminate the equality constraint without loss of generality by replacing it with two inequality constraints,  $g_j \leq \epsilon$  and  $-g_j \leq \epsilon$ , where  $\epsilon$  is some small number. In practice, numerical precision and the implementations of many methods make it desirable to distinguish between equality and inequality constraints.

---

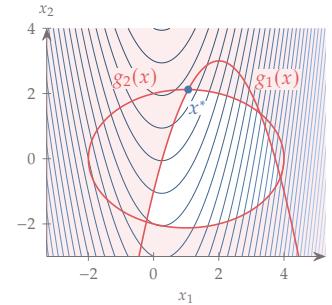
**Example 5.1:** Graphical solution of constrained problem.

Consider the following two-variable problem with quadratic objective and

constraint functions:

$$\begin{aligned} \text{minimize } & f(x_1, x_2) = x_1^2 - \frac{1}{2}x_1 - x_2 - 2 \\ \text{by varying } & x_1, x_2 \\ \text{subject to } & g_1(x_1, x_2) = x_1^2 - 4x_1 + x_2 + 1 \leq 0 \\ & g_2(x_1, x_2) = \frac{1}{2}x_1^2 + x_2^2 - x_1 - 4 \leq 0. \end{aligned} \quad (5.2)$$

We can plot the contours of the objective function and the constraint lines ( $g_1 = 0$  and  $g_2 = 0$ ), as shown in Fig. 5.1. We can see the feasible region defined by the two constraints and the approximate location of the minimum is evident by inspection. We are only able to visualize the contours for this problem because the functions can be evaluated quickly and because it has only two dimensions. If the functions were more expensive, we would not be able to afford the many evaluations needed to plot contours. If the problem had more dimensions, it would become difficult or impossible to fully visualize the functions and feasible space.



**Figure 5.1:** Graphical solution for constrained problem.

**Tip 5.2:** Do not mistake constraints for objectives.

Often a metric is posed an objective (usually with multiple objectives) when it is probably more appropriate as a constraint. This topic is discussed in more detail in Chapter 9.

The constrained problem formulation above does not distinguish between nonlinear and linear constraints or variable bounds. While it is advantageous to make this distinction, because some algorithms can take advantage of these differences, the methods introduced in this chapter will just assume general nonlinear functions.

**Tip 5.3:** Do not specify bounds as nonlinear constraints.

Bounds are a special category and the simplest form of a constraint:

$$\underline{x}_i \leq x_i \leq \bar{x}_i$$

Bounds are treated differently algorithmically and so should be specified as a bound constraint, rather than as a general nonlinear constraint. Some bounds will come from physical limitations on the engineering system. If not otherwise limited, the bounds should be sufficiently wide so as to not artificially constrain the problem. It is good practice to check your solution against your bounds to make sure you haven't artificially constrained the problem.

We need the Jacobian of the constraints throughout this chapter. The indexing order we use is:

$$[\nabla h]_{ij} = \begin{bmatrix} \nabla h_1^T \\ \vdots \\ \nabla h_{n_h}^T \end{bmatrix} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{n_h}}{\partial x_1} & \cdots & \frac{\partial h_{n_h}}{\partial x_{n_x}} \end{bmatrix} = \frac{\partial h_i}{\partial x_j}, \quad (5.3)$$

so this Jacobian is a matrix of size  $n_h \times n_x$ . The Jacobian of the inequality constraints uses the same index ordering.

## 5.2 Optimality Conditions

The optimality conditions for constrained optimization problems are not as straightforward as those for unconstrained optimization (Section 4.1.4). We begin with equality constraints because the mathematics and intuition are simpler, and then add inequality constraints. As in the case of unconstrained optimization, the optimality conditions for constrained problems are used not only for the termination criteria, but they are also used as the basis for optimization algorithms.

### 5.2.1 Equality Constraints

First, we review the optimality conditions for an unconstrained problem. For an unconstrained problem, we can take a first-order Taylor's series expansion of the objective function with some step  $p$  that is small enough that the second order term is negligible and write

$$f(x + p) \approx f(x) + \nabla f(x)^T p. \quad (5.4)$$

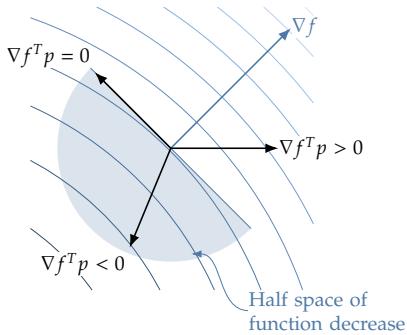
If  $x^*$  is a minimum point then every point in a small neighborhood must have a greater value,

$$f(x^* + p) \geq f(x^*). \quad (5.5)$$

Given the Taylor series expansion (5.4), the only way that this inequality can be satisfied is if

$$\nabla f(x^*)^T p \geq 0. \quad (5.6)$$

For a given  $\nabla f(x^*)$ , there are always an infinite number of directions along which the function decreases, which correspond to the halfspace shown in Fig. 5.2. If the problem is unconstrained then  $p$  can be in any direction and the only way to satisfy this inequality is if  $\nabla f(x^*) = 0$ .



**Figure 5.2:** The gradient  $f(x)$ , which is the direction of steepest function increase, splits the design space into two halves. Here we highlight the halfspace of directions that result in function decrease.

Therefore,  $\nabla f(x^*) = 0$  is a necessary condition for an unconstrained minimum.

Now consider the constrained case. The function increase condition (5.6) still applies, but  $p$  can only take *feasible* directions. To find the feasible directions, we can write a first-order Taylor series expansion for each equality constraint function as

$$h_j(x + p) \approx h_j(x) + \nabla h_j(x)^T p, \quad j = 1, \dots, n_h. \quad (5.7)$$

Again, the step size is assumed to be small enough so that the higher-order terms are negligible. Assuming we are at a feasible point, then  $h_j(x) = 0$  for all constraints  $j$ . To remain feasible, the step,  $p$ , must be such that the new point is also feasible, i.e.,  $h_j(x + p) = 0$  for all  $j$ . This implies that the feasibility of the new point requires

$$\nabla h_j(x)^T p = 0, \quad j = 1, \dots, n_h, \quad (5.8)$$

which means that *a direction is feasible when it is perpendicular to all equality constraint gradients*. Another way to look at this is that the feasible directions are in the intersection of the hyperplanes corresponding to the tangent of each constraint. These hyperplanes have at least one point in common by construction (the point we are evaluating,  $x$ ). Assuming distinct hyperplanes (linearly independent constraint gradients), their intersection defines a hyperplane with  $n_x - n_h$  degrees of freedom (see Fig. 5.3 for 2-D and 3-D illustrations).

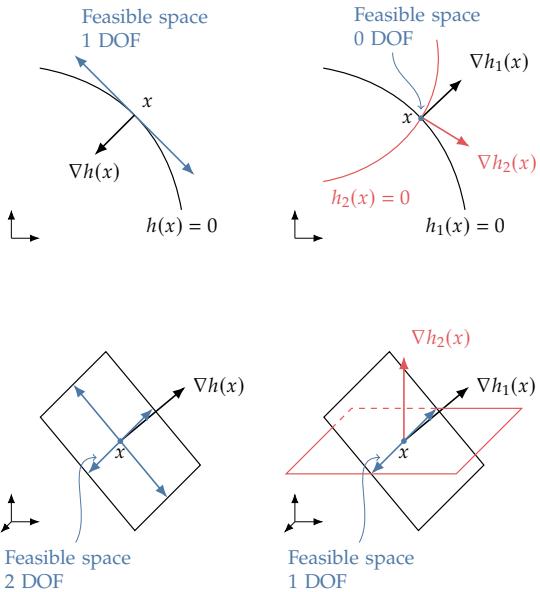
For a point to be a constrained minimum,

$$\nabla f(x^*)^T p \geq 0 \quad (5.9)$$

for all  $p$  such that

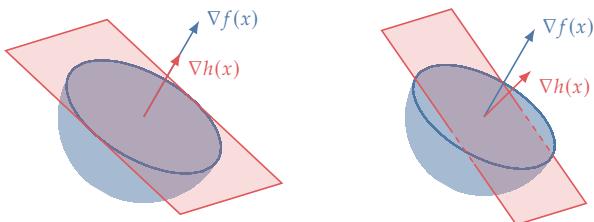
$$\nabla h_j(x^*)^T p = 0, \quad j = 1, \dots, n_h. \quad (5.10)$$

As previously mentioned, the feasible directions form a hyperplane in  $n_x$  dimensions. The intersection of this hyperplane with the half



**Figure 5.3:** Feasible spaces for 2-D examples with one constraint (upper left), and two constraints (upper right); 3-D examples with one constraint (lower left) and two constraints (lower right).

space containing all the descent directions ( $p$  such that  $\nabla f(x^*)^T p < 0$ ) must be zero. For this to happen, the only possibility to satisfy the inequality Eq. 5.9 is the case when it is zero. This is because a hyperplane in  $n_x$  dimensions includes directions in the descent halfspace of the same dimensions unless the hyperplane is perpendicular to  $\nabla f(x^*)$  (see Fig. 5.4 for an illustration in 3-D space).



**Figure 5.4:** All feasible directions must be contained in the hyperplane perpendicular to the gradient of the objective function so that there are no feasible descent directions. Here we show 3-D example with one constraint for a point satisfying optimality (left) and a point not satisfying optimality (right).

Another way of stating this condition is that the projection of  $\nabla f(x^*)$  onto all possible feasible directions must be zero. That is because if the projection were nonzero, there would be a feasible direction that is also a descent direction.

For the hyperplane defining the descent halfspace to align with the hyperplane of feasible directions, *the gradient of the objective must be a*

linear combination of the gradients of the constraints, i.e.,

$$\nabla f(x^*) = - \sum_{j=1}^{n_h} \lambda_j \nabla h_j(x^*), \quad (5.11)$$

where  $\lambda_j$  are called the *Lagrange multipliers*. There is a multiplier associated with each constraint. The sign is arbitrary for equality constraints, but will be significant later when dealing with inequality constraints and we choose the negative sign for consistency with this latter case.

To derive the full set of optimality conditions for constrained problems, it is convenient to define the *Lagrangian* function,

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T h(x), \quad (5.12)$$

where  $\lambda$  is the vector of Lagrange multipliers defined above, which are now unknown variables as well.<sup>\*</sup> The Lagrangian is defined such that its stationary points are candidate optima for the constrained problem. To find the stationary points, we can solve for  $\nabla \mathcal{L} = 0$ . Since  $\mathcal{L}$  is a function of both  $x$  and  $\lambda$  we need to set both partial derivatives equal to zero as follows,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_i} &= \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} = 0, \quad i = 1, \dots, n_x, \\ \frac{\partial \mathcal{L}}{\partial \lambda_j} &= h_j = 0, \quad j = 1, \dots, n_h. \end{aligned} \quad (5.13)$$

The first condition is the constrained optimality condition we explained previously in Eq. 5.11. The second condition enforces the equality constraints, which must still be enforced because the first condition could be satisfied at infeasible points.

With the Lagrangian function, we have transformed a constrained problem into an unconstrained problem by adding new variables,  $\lambda$ . A constrained problem of  $n_x$  design variables and  $n_h$  equality constraints was transformed into an unconstrained problem with  $n_x + n_h$  variables. Although you might be tempted to simply use the algorithms of Chapter 4 to solve the optimality conditions (5.13) for an unconstrained Lagrangian function, some modifications are needed in the algorithms to solve these problems effectively.

The optimality conditions above are first-order conditions that are necessary, but not sufficient. To make sure that a point is a constrained minimum, we also need to satisfy second-order conditions. For the unconstrained case, the Hessian of the objective function had to be

<sup>\*</sup>Despite our convention of reserving Greek symbols for scalars, we use  $\lambda$  to represent the vector of Lagrange multipliers as it is common usage.

positive definite. In the constrained case, we need to check the Hessian of the Lagrangian with respect to the design variables in the space of feasible directions. Therefore, the second order sufficient conditions are:

$$p^T [\nabla_{xx} \mathcal{L}] p > 0, \quad (5.14)$$

for all feasible  $p$ , so the projection of the curvature onto all feasible directions must be positive. The feasible directions are all directions  $p$  such that

$$\nabla h_j(x)^T p = 0, \quad j = 1, \dots, n_h. \quad (5.15)$$

That is, the feasible directions are in the null space of the Jacobian of the constraints.

---

**Example 5.4: Simple equality-constrained problem**

Consider the following constrained problem featuring a linear objective function and a quadratic equality constraint:

$$\begin{aligned} \text{minimize} \quad & f(x_1, x_2) = x_1 + 2x_2 \\ \text{subject to} \quad & h(x_1, x_2) = \frac{1}{4}x_1^2 + x_2^2 - 1 = 0. \end{aligned} \quad (5.16)$$

The Lagrangian for this problem is

$$\mathcal{L}(x_1, x_2, \lambda) = x_1 + 2x_2 + \lambda \left( \frac{1}{4}x_1^2 + x_2^2 - 1 \right). \quad (5.17)$$

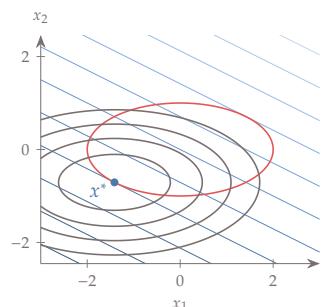
Differentiating this to get the first-order optimality conditions,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_1} &= 1 + \frac{1}{2}\lambda x_1 = 0 \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2 + 2\lambda x_2 = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= \frac{1}{4}x_1^2 + x_2^2 - 1 = 0. \end{aligned} \quad (5.18)$$

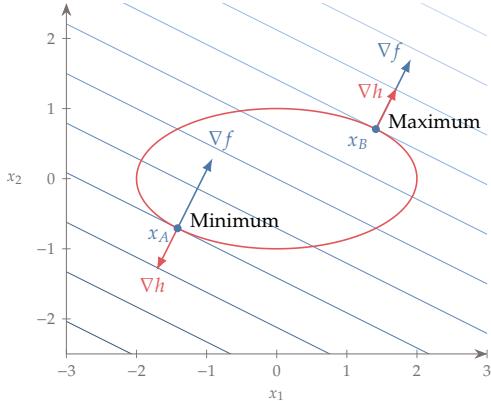
Solving these three equations for the three unknowns  $(x_1, x_2, \lambda)$ , we obtain two possible solutions:

$$\begin{aligned} x_A &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -\sqrt{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}, \quad \lambda_A = \sqrt{2}, \\ x_B &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}, \quad \lambda_B = -\sqrt{2}. \end{aligned} \quad (5.19)$$

These two points are shown in Fig. 5.5, together with the objective and constraint gradients, which align with each other as expected. To determine if



**Figure 5.6:** The minimum of the Lagrangian function with the optimum Lagrange multiplier value ( $\sigma = \sqrt{2}$ ) is the constrained minimum of the original problem.



**Figure 5.5:** Two points satisfy the first-order KKT conditions; one is a constrained minimum and the other is a constrained maximum.

either of these points is a minimum, we check the second-order conditions by evaluating the Hessian of the Lagrangian,

$$\nabla_{xx} \mathcal{L} = \begin{bmatrix} \frac{1}{2}\lambda & 0 \\ 0 & 2\lambda \end{bmatrix}. \quad (5.20)$$

The Hessian is only positive definite for the case where  $\lambda_A = \sqrt{2}$  and therefore  $x_A$  is an optimum. Recall that the Hessian only needs to be positive definite in the feasible directions, but here we can easily show that it is positive or negative definite in all possible directions. The Hessian is negative definite for  $x_B$ , so this is not a minimum; instead, it is a maximum in this case.

### 5.2.2 Inequality Constraints

We can reuse the optimality conditions we derived for equality constraints for the inequality constrained problems. The key insight is that the equality conditions apply to inequality constraints that are active, while inactive constraints can be ignored. Recall that for a general inequality constraint  $g_j(x) \leq 0$ , constraint  $j$  is said to be *active* if  $g_j(x^*) = 0$  and *inactive* if  $g_j(x^*) < 0$ .

As before, if  $x^*$  is an optimum, any small enough step  $p$  from the optimum must result in a function increase. Based on the Taylor series expansion (5.4), we get the condition

$$\nabla f(x^*)^T p \geq 0, \quad (5.21)$$

which is the same as for the equality constrained case.

To consider the constraints, we use the same linearization of the constraint (5.7), but now we enforce an inequality to get

$$g_j(x + p) \approx g_j(x) + \nabla g_j(x)^T p \leq 0, \quad j = 1, \dots, n_g. \quad (5.22)$$

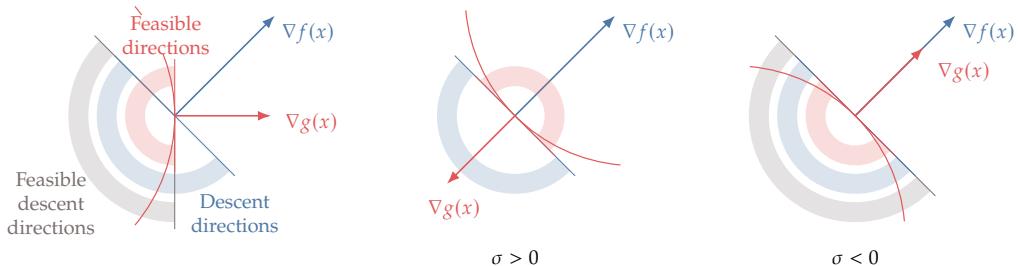
For a given candidate point that satisfies the constraints, there are two possibilities to consider for each constraint: whether the constraint is inactive ( $g_j(x) < 0$ ) or active ( $g_j(x) = 0$ ). If a given constraint is inactive, then we do not need to add any conditions because we can take a step,  $p$ , in any direction and remain feasible as long as the step is small enough. If a given constraint is active, then we can treat it as an equality constraint.

Thus, the optimality conditions derived for the equality constrained case can be reused here, but with a crucial modification. First, the requirement that the gradient of the objective is a linear combination of the gradients of the constraints, only needs to consider the active constraints. This can be written as

$$\nabla f(x^*) = - \sum_{j=1}^{n_{g,\text{active}}} \sigma_j \nabla g_j(x^*), \quad (5.23)$$

where  $\sigma$  is the Lagrange multiplier for the inequality constraints, and the summation occurs only over the active constraints.

Second, the sign of the Lagrange multipliers is now significant, and must be positive (by our convention) for a feasible optimum. This is because the feasible space is no longer a hyperplane, but the intersection of the halfspaces defined by the constraints. An illustration of a 2-D case with one constraint is shown in Fig. 5.7.



We need to include all inequality constraints in the optimality conditions because we do not know in advance which constraints are active. To do this, we replace the inequality constraints  $g_k \leq 0$  with the equality constraints:

$$g_k + s_k^2 = 0, \quad k = 1, \dots, n_g \quad (5.24)$$

where  $s_k$  is a new unknown associated with each inequality constraint called a *slack variable*. The slack variable is squared to ensure it is positive, so that  $g_k$  is nonpositive and thus feasible for any  $s_k$ . The significance of the slack variable is that when  $s_k = 0$ , then the corresponding inequality

**Figure 5.7:** Constrained minimum conditions for 2-D case with one inequality constraint. The objective function gradient must be parallel and have opposite directions (corresponding to a positive Lagrange multiplier) so that there are no feasible descent directions.

constraint is active ( $g_k = 0$ ), and when  $s_k \neq 0$ , the corresponding constraint is inactive.

The Lagrangian including both equality and inequality constraints is then

$$\mathcal{L}(x, \lambda, \sigma, s) = f(x) + \lambda^T h(x) + \sigma^T (g(x) + s^2), \quad (5.25)$$

where  $\sigma$  are the Lagrange multipliers associated with the inequality constraints.

Similarly to the equality constrained case, we seek a stationary point for the Lagrangian, but now we have additional unknowns: the inequality Lagrange multipliers and the slack variables. Taking partial derivatives with respect to the design variables and setting them to zero, we obtain

$$\nabla_x \mathcal{L} = 0 \Rightarrow \frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} + \sum_{k=1}^{n_g} \sigma_k \frac{\partial g_k}{\partial x_i} = 0, \quad i = 1, \dots, n_x \quad (5.26)$$

This criteria is the same as before, but with additional Lagrange multipliers and constraints. Taking the derivatives with respect to the equality Lagrange multipliers,

$$\nabla_\lambda \mathcal{L} = 0 \Rightarrow \frac{\partial \mathcal{L}}{\partial \lambda_j} = h_j = 0, \quad j = 1, \dots, n_h, \quad (5.27)$$

which enforced the equality constraints as before. Taking derivatives with respect to the inequality Lagrange multipliers, we get

$$\nabla_\sigma \mathcal{L} = 0 \Rightarrow \frac{\partial \mathcal{L}}{\partial \sigma_k} = g_k + s_k^2 = 0 \quad k = 1, \dots, n_g, \quad (5.28)$$

which enforces the inequality constraints. Finally, differentiating the Lagrangian with respect to the slack variables, we obtain

$$\nabla_s \mathcal{L} = 0 \Rightarrow \frac{\partial \mathcal{L}}{\partial s_k} = 2\sigma_k s_k = 0, \quad k = 1, \dots, n_g, \quad (5.29)$$

which is called the *complementarity condition*. This condition helps us to distinguish the active constraints from the inactive ones. For each inequality constraint, either the Lagrange multiplier is zero (which means that the constraint is inactive), or the slack variable is zero (which means that the constraint is active). Unfortunately, this condition introduces a combinatorial problem whose complexity grows exponentially with the number of inequality constraints, since the number of combinations of active versus inactive constraints is  $2^{n_g}$ .

These requirements are called the Karush–Kuhn–Tucker (KKT) conditions and are summarized below:

$$\begin{aligned} \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} + \sum_{k=1}^{n_g} \sigma_k \frac{\partial g_k}{\partial x_i} &= 0, \quad i = 1, \dots, n_x \\ h_j &= 0, \quad j = 1, \dots, n_h \\ g_k + s_k^2 &= 0, \quad k = 1, \dots, n_g \\ \sigma_k s_k &= 0, \quad k = 1, \dots, n_g \\ \sigma_k &\geq 0, \quad k = 1, \dots, n_g \end{aligned} \tag{5.30}$$

The last addition, that the Lagrange multipliers associated with the inequality constraints must be nonnegative, was implicit in the way we defined the Lagrangian but is now made explicit. As shown in Fig. 5.7, the Lagrange multiplier for an inequality constraint must be positive otherwise there is a direction that is feasible and would decrease the objective function.

The equality and inequality constraints are often lumped together for convenience, since the expression for the Lagrangian follows the same form for both cases. As in the equality constrained case, these conditions are necessary but not sufficient. We still need to require that the Hessian of the Lagrangian be positive definite in all feasible directions, as stated in Eq. 5.14. In the inequality case, the feasible directions are all in the null space of the Jacobian of the equality constraints *and* active inequality constraints, that is,

$$\begin{aligned} \nabla h_j(x)^T p &= 0, \quad j = 1, \dots, n_h, \\ \nabla g_i(x)^T p &= 0, \quad \text{for all } i \text{ in active set.} \end{aligned} \tag{5.31}$$

---

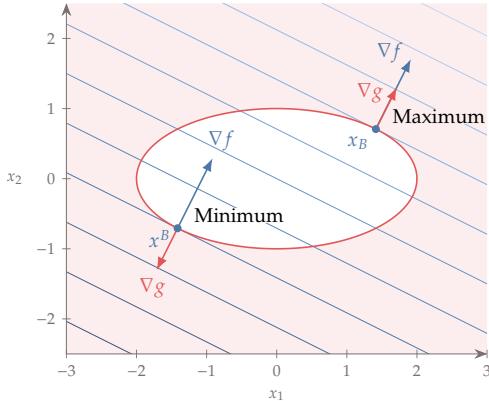
**Example 5.5:** Simple problem with one inequality constraint

Consider a variation of the simple problem (Ex. 5.4), where the equality is replaced by an inequality as follows:

$$\begin{aligned} \text{minimize} \quad & f(x_1, x_2) = x_1 + 2x_2 \\ \text{subject to} \quad & g(x_1, x_2) = \frac{1}{4}x_1^2 + x_2^2 - 1 \leq 0. \end{aligned} \tag{5.32}$$

The Lagrangian for this problem is

$$\mathcal{L}(x_1, x_2, \sigma, s) = x_1 + 2x_2 + \sigma \left( \frac{1}{4}x_1^2 + x_2^2 - 1 + s^2 \right). \tag{5.33}$$



**Figure 5.8:** Inequality problem with linear objective and feasible space within an ellipse.

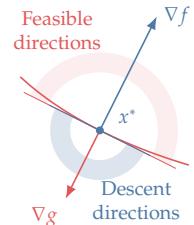
Differentiating this with respect to all the variables to get the first-order optimality conditions,

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_1} &= 1 + \frac{1}{2}\sigma x_1 = 0, \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2 + 2\sigma x_2 = 0, \\ \frac{\partial \mathcal{L}}{\partial \sigma} &= \frac{1}{4}x_1^2 + x_2^2 - 1 = 0, \\ \frac{\partial \mathcal{L}}{\partial s} &= 2\sigma s = 0.\end{aligned}\tag{5.34}$$

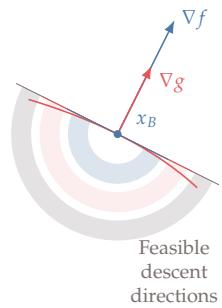
Starting with the last equation, there are two possibilities:  $s = 0$  (meaning the constraint is active) and  $\sigma = 0$  (meaning the constraint is not active). However, we can see that setting  $\sigma = 0$  in either of the two first equations does not yield a solution. Assuming that  $s = 0$  and  $\sigma \neq 0$ , we can solve the equations to obtain the same points as in Ex. 5.4:

$$x_A = \begin{bmatrix} x_1 \\ x_2 \\ \sigma \end{bmatrix} = \begin{bmatrix} -\sqrt{2} \\ -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}, x_B = \begin{bmatrix} x_1 \\ x_2 \\ \sigma \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ \frac{\sqrt{2}}{2} \\ -\sqrt{2} \end{bmatrix}.\tag{5.35}$$

There are the same critical points as in the equality constrained case of Ex. 5.4. However, now the sign of the Lagrange multiplier has meaning. According to the KKT conditions, the Lagrange multiplier has to be non-negative. Only  $x_A$  satisfies this condition and therefore there is no descent direction that is feasible, as shown in Fig. 5.9. The Hessian of the Lagrangian at this point is the same as in Ex. 5.4, where we showed it is positive definite. Therefore,  $x_A$  is a minimum. Unlike the equality-constrained problem, we did not need to check the Hessian of point  $x_B$  because the Lagrange multiplier is negative, and as a consequence there are feasible descent directions, as shown in Fig. 5.10.



**Figure 5.9:** At the minimum there the Lagrange multiplier is positive and there is no descent direction that is feasible.



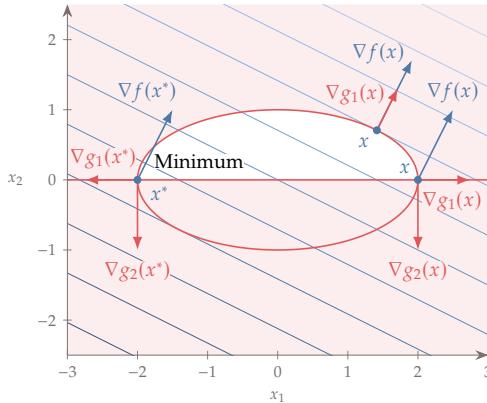
**Figure 5.10:** At this critical point, the Lagrange multiplier is negative and all descent directions are feasible, so this point is not a minimum.

**Example 5.6:** Simple problem with two inequality constraints

Consider a variation of Ex. 5.5, where we add one more inequality as follows

$$\begin{aligned} \text{minimize } & f(x_1, x_2) = x_1 + 2x_2 \\ \text{subject to } & g_1(x_1, x_2) = \frac{1}{4}x_1^2 + x_2^2 - 1 \leq 0, \\ & g_2(x_2) = -x_2 \leq 0. \end{aligned} \quad (5.36)$$

The feasible region is the top half of the ellipse, as shown in Fig. 5.11. The



**Figure 5.11:** Only one point satisfies the first-order KKT conditions.

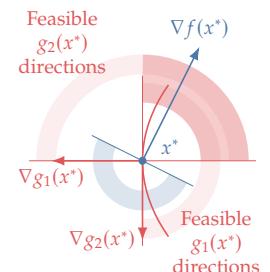
Lagrangian for this problem is

$$\mathcal{L}(x_1, x_2, \sigma, s) = x_1 + 2x_2 + \sigma_1 \left( \frac{1}{4}x_1^2 + x_2^2 - 1 + s_1^2 \right) + \sigma_2 (-x_2 + s_2^2). \quad (5.37)$$

Differentiating this with respect to all the variables to get the first-order optimality conditions,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_1} &= 1 + \frac{1}{2}\sigma_1 x_1 = 0, \\ \frac{\partial \mathcal{L}}{\partial x_2} &= 2 + 2\sigma_1 x_2 - \sigma_2 = 0, \\ \frac{\partial \mathcal{L}}{\partial \sigma_1} &= \frac{1}{4}x_1^2 + x_2^2 - 1 + s_1^2 = 0, \\ \frac{\partial \mathcal{L}}{\partial \sigma_2} &= -x_2 + s_2^2 = 0, \\ \frac{\partial \mathcal{L}}{\partial s_1} &= 2\sigma_1 s_1 = 0, \\ \frac{\partial \mathcal{L}}{\partial s_2} &= 2\sigma_2 s_2 = 0. \end{aligned} \quad (5.38)$$

We now have two complementarity conditions, which yield the four potential combinations listed in Ex. 5.6. Assuming that both constraints are active yields two possible solutions corresponding to two different Lagrange multipliers. According to the KKT conditions, the Lagrange multiplier for an active inequality constraint has to be positive, so only the solution with  $\sigma_1 = 1$  is a candidate



**Figure 5.12:** At the minimum, the intersection of the feasible directions and descent directions is null, so there is no feasible descent direction.

**Table 5.1:** Two inequality constraints yield four potential combinations.

Assumption	Meaning	$x_1$	$x_2$	$\sigma_1$	$\sigma_2$	$s_1$	$s_2$
$s_1 = 0, s_2 = 0$	Both constraints are active	-2	0	1	2	0	0
		2	0	-1	2	0	0
$\sigma_1 = 0, \sigma_2 = 0$	Neither constraint is active	-	-	-	-	-	-
$s_1 = 0, \sigma_2 = 0$	Only constraint 1 is active	$\sqrt{2}$	$\frac{\sqrt{2}}{2}$	$-\sqrt{2}$	0	0	$2^{-\frac{1}{4}}$
$\sigma_1 = 0, s_2 = 0$	Only constraint 2 is active	-	-	-	-	-	-

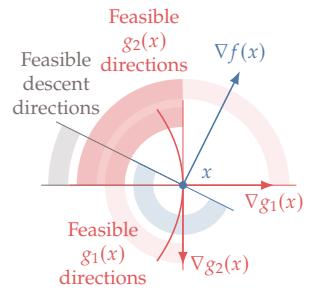
for a minimum (see Fig. 5.12). The Hessian of the Lagrangian is identical to the previous example and is positive definite when  $\sigma_1$  is positive. The other solution corresponds to  $x = (2, 0)$ , where there is a cone of descent directions that is feasible, as shown in Fig. 5.13. Assuming that neither constraint is active yields 1 = 0 for the first optimality condition, so this situation is not possible. Assuming that the first constraint is active yields the solution corresponding to the maximum that we already found in Ex. 5.5 and shown in Fig. 5.10. Finally, assuming that only the second constraint is active yields no candidate point.

While these examples can be solved analytically, they are the exception rather than the rule. The KKT conditions quickly become challenging to solve analytically (try solving Ex. 5.1). Furthermore, engineering problems usually involve functions that are defined by models with implicit equations, which are impossible to solve analytically. The reason we include these analytic examples is to better understand the KTT conditions. For the rest of the chapter, we focus on numerical methods, which are necessary for the vast majority of practical problems.

### 5.2.3 Meaning of the Lagrange Multipliers

A useful way to think of Lagrange multipliers is that they are the sensitivity of the optimal objective  $f(x^*)$  to the value of the corresponding constraints. Here we will explain why that is the case and how it provides design intuition.

When a constraint is inactive, the corresponding Lagrange multiplier is zero. This indicates that changing the value of an inactive constraint does not affect the optimum, which is intuitive. This is only valid to first order because the KKT conditions are based on the linearization of the objective and constraint functions. Therefore, small changes are



**Figure 5.13:** At this point, there is a cone of descent directions that is also feasible, so it is not a minimum.

assumed; an inactive constraint could be made active by changing its value by a large enough amount.

Consider taking the derivative of Eq. 5.25 with respect to the  $i^{\text{th}}$  inequality constraint ( $g_i$ ):

$$\frac{\partial \mathcal{L}}{\partial g_i} = \sigma_i \quad (5.39)$$

With a similar form for the Lagrange multiplier for the equality constraints. Thus, to first order, the Lagrange multipliers tell us how much the Lagrangian would be expected to change by changing the constraint by a unit amount. This has practical value because it tells us how much of an improvement can be expected if a given constraint is relaxed. The Lagrange multipliers quantify how much the corresponding constraints drive the design.

### 5.3 Penalty Methods

The concept behind penalty methods is intuitive; to transform a constrained problem into an unconstrained one by adding a penalty to the objective function when constraints are violated. As mentioned in the introduction to this chapter, penalty methods are no longer used directly in gradient-based optimization algorithms, because it is difficult to get them to converge to the true solution. However, these methods are still useful to discuss because: 1) they are simple and thus ease the transition into understanding constrained optimization; 2) though not effective for gradient-based optimization, they are still useful in some constrained gradient-free methods, as will be discussed in Chapter 7; 3) they can be useful as merit functions in line search algorithms, as discussed in Section 5.6.

The penalized function can be written as

$$F(x) = f(x) + \mu P(x), \quad (5.40)$$

where  $P(x)$  is a penalty function and the scalar  $\mu$  is a penalty parameter. This is similar in form to the Lagrangian, but one difference is that a value for  $\mu$  is fixed in advance instead of solved for.

We can use the unconstrained optimization techniques to minimize  $F(x)$ . However, instead of just solving a single optimization problem, penalty methods usually solve a sequence of problems with different values of  $\mu$  to get closer to the actual constrained minimum. We will see shortly why we need to solve a sequence of problems rather than just one problem.

Different forms for  $P(x)$  can be used, leading to different penalty methods. There are two main types of penalty functions: exterior

penalties, which impose a penalty only when constraints are violated, and interior penalty functions, which impose a penalty that increases as a constraint is approached.

### 5.3.1 Exterior Penalty Methods

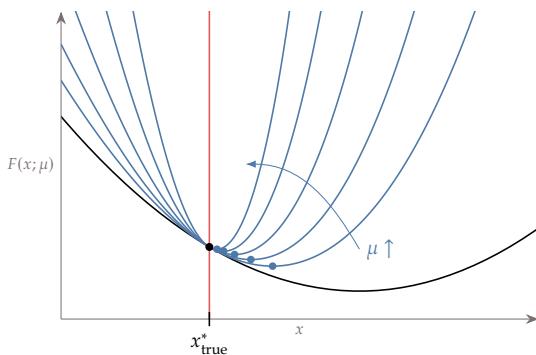
Of the many possible exterior penalty methods, we focus on two of the most popular ones: quadratic penalties and the augmented Lagrangian method. Quadratic penalties are continuously differentiable and simple to implement, but suffer from numerical ill-conditioning. The augmented Lagrangian method is more sophisticated; it is based on the quadratic penalty but adds terms that improve the numerical properties. Many other penalties are possible, such as L1-norms, which are often used when continuous differentiability is not necessary.

#### Quadratic Penalty Method

For equality-constrained problems the quadratic penalty method takes the form,

$$F(x; \mu) = f(x) + \frac{\mu}{2} \sum_i h_i(x)^2. \quad (5.41)$$

The motivation for a quadratic penalty is that it is simple and results in a function that is continuously differentiable. The factor of one half is unnecessary, but is included by convention as it eliminates the extra factor of two when taking derivatives. The penalty is nonzero unless the constraints are satisfied ( $h_i = 0$ ), as desired.



**Figure 5.14:** Quadratic penalty for an equality constrained 1-D problem. The minimum of the penalized function (blue dots) approaches the true constrained minimum (black circle) as the penalty parameter  $\mu$  increases.

The value of the penalty parameter  $\mu$  must be chosen carefully. Mathematically, we recover the exact solution to the constrained problem only as  $\mu$  tends to infinity (see Fig. 5.14). However, starting with a large value for  $\mu$  is not practical. This is because the larger the value of  $\mu$ , the larger the Hessian condition number, which corresponds to

the curvature varying greatly with direction (as an example, think of a quadratic function where the level curves are highly skewed). This behavior makes the problem difficult to solve numerically.

To solve the problem more effectively, we begin with a small value of  $\mu$  and solve the unconstrained problem. We then increase  $\mu$  and solve the new unconstrained problem, using the previous solution as the starting point for this problem. This process is repeated until the optimality conditions are satisfied (or some other approximate convergence criteria are satisfied), as outlined in Alg. 5.7. By gradually increasing  $\mu$  and reusing the solution from the previous problem, we avoid some of the ill-conditioning issues. Thus, the original constrained problem is transformed into a sequence of unconstrained optimization problems.

---

**Algorithm 5.7:** Exterior penalty method

---

**Inputs:** $x_0$ : Starting point $\mu_0 > 0$ : Initial penalty parameter $\rho > 1$ : Penalty increase factor**Outputs:** $x^*$ : Optimal point $f(x^*)$ : Corresponding function value $k = 0$ **while** not converged **do** $x_k^* \leftarrow \text{minimize } F(x_k; \mu_k) \text{ with respect to } x_k$ *Increase penalty parameter\** $\mu_{k+1} = \rho \mu_k$ *Update starting point for next optimization* $x_{k+1} = x_k^*$  $k = k + 1$ **end while**


---

There are three potential issues with the approach outlined in Alg. 5.7. If the starting value for  $\mu$  is too low, then the penalty might not be enough to overcome a function that is unbounded from below, and the penalized function has no minimum.

The second issue is that we cannot practically approach  $\mu \rightarrow \infty$ , so the solution to the problem is always slightly infeasible. By comparing the optimality condition of the constrained problem,  $\nabla_x \mathcal{L} = 0$ , and the optimality of the penalized function,  $\nabla_x F = 0$ , we can show that for

---

\*  $\rho$  may range from a conservative value (1.2) to an aggressive value (10), depending on the problem.

each constraint  $i$ ,

$$h_i \approx \frac{\lambda_i^*}{\mu}. \quad (5.42)$$

Since  $h_i = 0$  for the exact optimum,  $\mu$  must be made large to satisfy the constraints.

The third issue has to do with the curvature of the penalized function, which is directly proportional to  $\mu$ . The added curvature is added in a direction perpendicular to the constraints, which makes the Hessian of the penalized function increasingly ill-conditioned as  $\mu$  increases. When using Newton or quasi-Newton methods, such ill-conditioning causes numerical difficulties.

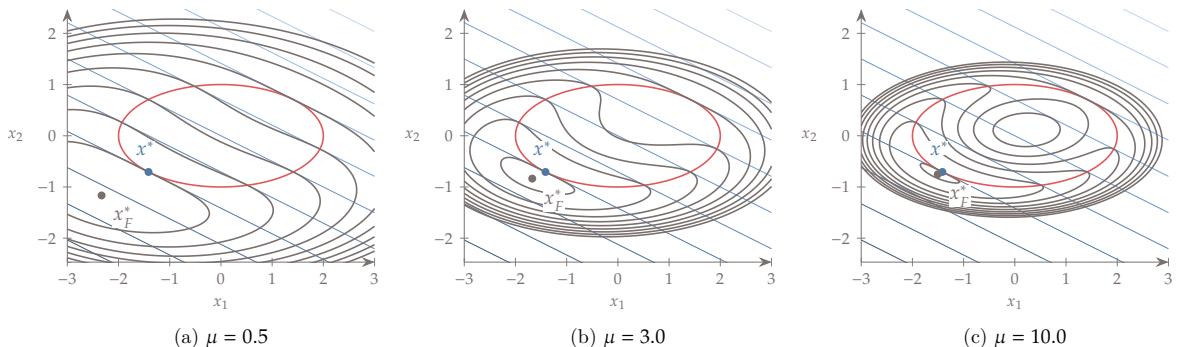
---

**Example 5.8:** Quadratic penalty for equality constrained problem

Consider the equality constrained problem from Ex. 5.4. The penalized function for that case is

$$F(x; \mu) = x_1 + 2x_2 + \frac{\mu}{2} \left( \frac{1}{4}x_1^2 + x_2^2 - 1 \right)^2. \quad (5.43)$$

This function is shown in Fig. 5.15 for different values of the penalty parameter  $\mu$ . The penalty is active for all points that are infeasible, but the minimum of the penalized function does not coincide with the constrained minimum of the original problem. The penalty parameter needs to be increased for the minimum of the penalized function to approach the correct solution, but this results in a highly nonlinear function.



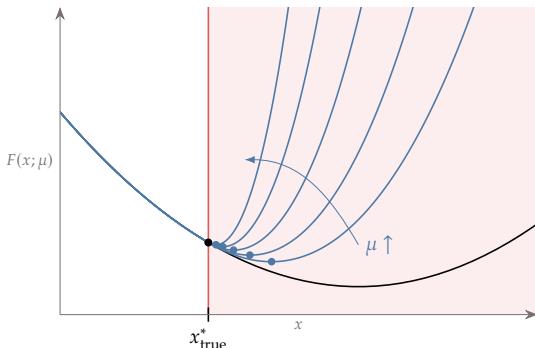
The approach discussed so far handles only equality constraints, but we can extend it to handle inequality constraints: Instead of adding a penalty to both sides of the constraints, we just add the penalty when the inequality constraint is violated (i.e., when  $g_j(x) > 0$ ). This behavior

**Figure 5.15:** Quadratic penalty for one equality constraint. The minimum of the penalized function approaches the constrained minimum as the penalty parameter increases.

can be achieved by defining a new penalty function as

$$F(x; \mu) = f(x) + \frac{\mu}{2} \sum_j \max[0, g_j(x)]^2. \quad (5.44)$$

The only difference relative to the equality constraint penalty shown in Fig. 5.14 is that the penalty is removed on the feasible side of the inequality constraint, as shown in Fig. 5.16.



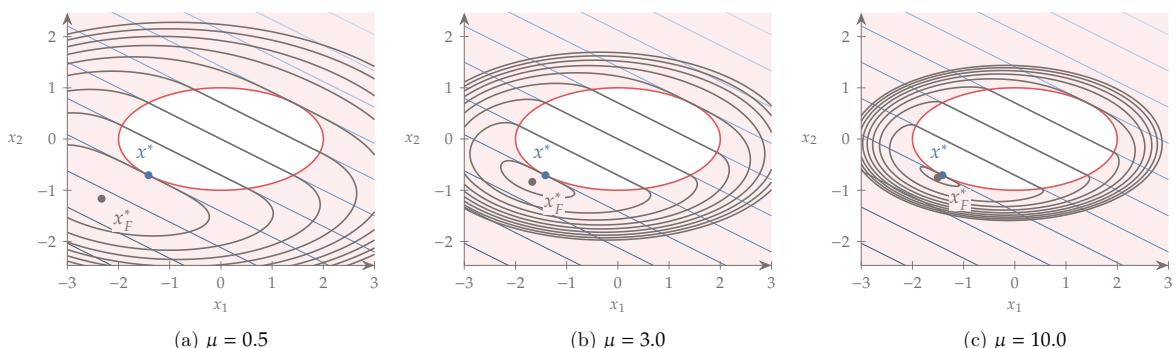
**Figure 5.16:** Quadratic penalty for an inequality constrained 1-D problem. The minimum of the penalized function approaches the constrained minimum from the infeasible side.

#### Example 5.9: Quadratic penalty for inequality constrained problem

Consider the equality constrained problem from Ex. 5.5. The penalized function for that case is

$$F(x; \mu) = x_1 + 2x_2 + \frac{\mu}{2} \max \left[ 0, \frac{1}{4}x_1^2 + x_2^2 - 1 \right]^2. \quad (5.45)$$

This function is shown in Fig. 5.17 for different values of the penalty parameter  $\mu$ . The contours of the feasible region inside the ellipse coincide with the original function contours, but outside the feasible region, the contours change to create a function whose minimum approaches the exact constrained minimum as the penalty parameters is increased.



**Figure 5.17:** Quadratic penalty for one inequality constraint. The minimum of the penalized function approaches the constrained minimum from the infeasible side.

The inequality quadratic penalty can be used together with the quadratic penalty for equality constraints if we need to handle both types of constraints. The two penalty parameters can be incremented in lockstep or independently.

**Tip 5.10:** Scaling is also important for constrained problems.

The same considerations on scaling discussed in Chapter 4 are just as important for constrained problems. As a rule of thumb, all constraints should be of order one.

## Augmented Lagrangian

As explained above, the quadratic penalty method requires a large value of  $\mu$  for constraint satisfaction, but the large  $\mu$  degrades the numerical conditioning. The augmented Lagrangian method alleviates this dilemma by adding the quadratic penalty to the Lagrangian instead of just adding it to the function. The augmented Lagrangian function for equality constraints is:

$$F(x, \lambda; \mu) = f(x) + \sum_j^{n_h} \lambda_j h_j(x) + \frac{\mu}{2} \sum_j^{n_h} h_j(x)^2, \quad (5.46)$$

Inequality constraints can be included in a similar way using the maximum function of Eq. 5.44 and considering only the active or violated constraints in the second term. Unfortunately, the Lagrange multipliers cannot be solved for in a penalty approach and so we need some way to estimate them.

To obtain an estimate of the Lagrange multipliers, we can compare the optimality conditions for the augmented Lagrangian,

$$\nabla_x F(x, \lambda; \mu) = \nabla f(x) + \sum_j^{n_h} [\lambda_j + \mu h_j(x)] \nabla h_j = 0 \quad (5.47)$$

to those of the actual Lagrangian,

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = \nabla f(x^*) + \sum_j^{n_h} \lambda_j^* \nabla h_j(x^*) = 0, \quad (5.48)$$

which suggests the approximation

$$\lambda_j^* \approx \lambda_j + \mu h_j. \quad (5.49)$$

Therefore, we update the vector of Lagrange multipliers based on the current estimate of the Lagrange multipliers and constraint values using

$$\lambda_{k+1} = \lambda_k + \mu_k h(x_k) \quad (5.50)$$

The complete algorithm is shown in Alg. 5.11.

---

**Algorithm 5.11:** Augmented Lagrangian penalty method
 

---

**Inputs:**

- $x_0$ : Starting point
- $\lambda_0 = 0$ : Initial Lagrange multiplier
- $\mu_0 > 0$ : Initial penalty parameter
- $\rho > 1$ : Penalty increase factor

**Outputs:**

- $x^*$ : Optimal point
  - $f(x^*)$ : Corresponding function value
- 

```

 $k = 0$ 
while not converged do
   $x_k^* \leftarrow$  minimize  $F(x_k, \lambda_k; \mu_k)$  with respect to  $x_k$ 
   $\lambda_{k+1} = \lambda_k + \mu_k h(x_k)$  Update Lagrange multipliers
   $\mu_{k+1} = \rho \mu_k$  Increase penalty parameter
   $x_{k+1} = x_k^*$  Update starting point for next optimization
   $k = k + 1$ 
end while

```

---

The reason why this approach is an improvement on the plain quadratic penalty is because by updating the Lagrange multiplier estimates at every iteration, we obtain more accurate solutions without having to increase  $\mu$  too much. We can see this by comparing the augmented Lagrangian approximation to the constraints obtained from Eq. 5.49,

$$h_i \approx \frac{1}{\mu}(\lambda_i^* - \lambda_i) \quad (5.51)$$

with the corresponding approximation in the quadratic penalty method,

$$h_i \approx \frac{\lambda_i^*}{\mu}. \quad (5.52)$$

To drive the constraints to zero, the quadratic penalty relies solely on increasing  $\mu$  in the denominator. However, with the augmented Lagrangian, we also have control on the numerator through the Lagrange multiplier estimate. If the estimate is reasonably close to the

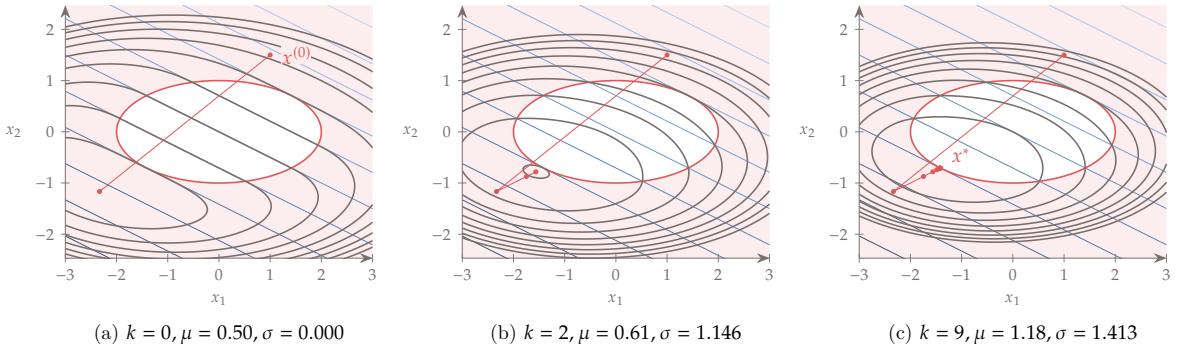
true Lagrange multiplier then the numerator will become small for modest values of  $\mu$ . Thus, the augmented Lagrangian can provide a good solution for  $x^*$  while avoiding the ill-conditioning issues of the quadratic penalty.

**Example 5.12:** Augmented Lagrangian for inequality constrained problem

Consider the equality constrained problem from Ex. 5.5. Assuming the inequality constraint is active, the augmented Lagrangian (Eq. 5.46) for that problem is

$$F(x; \mu) = x_1 + 2x_2 + \sigma \left( \frac{1}{4}x_1^2 + x_2^2 - 1 \right) + \frac{\mu}{2} \left( \frac{1}{4}x_1^2 + x_2^2 - 1 \right)^2. \quad (5.53)$$

Applying Alg. 5.11, starting with  $\mu = 0.5$  and using  $\rho = 1.1$ , we get the iterations



shown in Fig. 5.18. Compared to the quadratic penalty in Ex. 5.9, the penalized function is much better conditioned, thanks to the term associated with the Lagrange multiplier. The minimum of the penalized function eventually becomes the minimum of the constrained problem without the need for a large penalty parameter.

**Figure 5.18:** Augmented La-  
grangian applied to inequality  
constrained problem.

### 5.3.2 Interior Penalty Methods

Interior penalty methods work the same way as exterior penalty methods—they transform the constrained problem into a series of unconstrained problems. The main difference with interior penalty methods is that they seek to always maintain feasibility: Instead of adding a penalty only when constraints are violated, they add a penalty as the constraint is approached from the feasible region. This type of penalty is particularly desirable if the objective function is ill-defined outside the feasible region. These methods are called “interior” because

the iteration points remain on the interior of the feasible region. They are also referred to as barrier methods because the penalty function acts as a barrier preventing iterates from leaving the feasible region.

One possible interior penalty function to enforce  $g(x) \leq 0$  is the inverse function (top of Fig. 5.19),

$$P(x) = \sum_j^{n_g} -\frac{1}{g_j(x)}, \quad (5.54)$$

where  $P(x) \rightarrow \infty$  as  $c_i(x) \rightarrow 0^-$ .

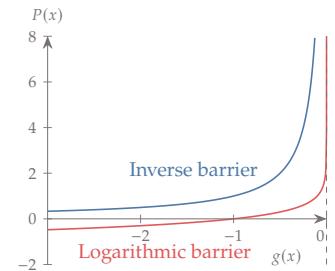
A more popular interior penalty function is the logarithmic barrier (bottom of Fig. 5.19),

$$P(x) = \sum_j^{n_g} -\log(-g_j(x)), \quad (5.55)$$

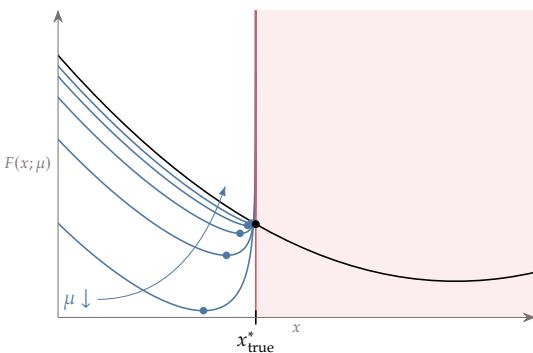
which also approaches infinity as the constraint tends to zero from the feasible side. The penalty function is then,

$$F(x; \mu) = f(x) - \mu \sum_j^{n_g} \log(-g_j(x)). \quad (5.56)$$

Like exterior methods, interior methods must also solve a sequence of unconstrained problems but with  $\mu \rightarrow 0$  (see Alg. 5.13). As the penalty parameter is decreased, the region across which the penalty acts decreases making it sharper and more like a barrier as shown in Fig. 5.20.



**Figure 5.19:** Two different interior barrier functions.



**Figure 5.20:** Logarithmic barrier penalty for an inequality constrained 1-D problem. The minimum of the penalized function (blue circles) approaches the true constrained minimum (black circle) as the penalty parameter  $\mu$  decreases.

---

Algorithm 5.13: Interior penalty method

**Inputs:**

$x_0$ : Starting point

$\mu_0 > 0$ : Initial penalty parameter

$\rho < 1$ : Penalty decrease factor

### Outputs:

$x^*$ : Optimal point

$f(x^*)$ : Corresponding function value

$k = 0$

**while** not converged **do**

$x_k^* \leftarrow$  minimize  $F(x_k; \mu_k)$  with respect to  $x_k$

*Decrease penalty parameter*

$\mu_{k+1} = \rho \mu_k$

*Update starting point for next optimization*

$x_{k+1} = x_k^*$

$k = k + 1$

**end while**

The methodology is essentially the same as is described in Alg. 5.7, but with a decreasing penalty parameter. One major weakness of the method is that the penalty function is not defined for infeasible points so a feasible starting point must be provided. For some problems, providing a feasible starting point may be difficult or practically impossible. To prevent the algorithm from going infeasible when starting from a feasible point, the line search must be safeguarded. For the logarithmic barrier, this can be done by checking the values of the constraints and backtracking if any of them is greater than or equal to zero.

Both interior and exterior penalties are shown for a two-dimensional function in Fig. 5.21. The exterior penalty leads to solutions that are slightly infeasible, while an interior penalty leads to a feasible solution but underpredicts the objective.

Another weakness is that, similarly to the exterior penalty methods, the Hessian becomes ill-conditioned as the penalty parameter tends to zero.<sup>64</sup> There are augmented and modified barrier approaches that can avoid the ill-conditioning issue (and other methods that remain ill-conditioned but can still be solved reliably, albeit inefficiently).<sup>65</sup> However, these methods have been superseded by modern interior point methods discussed in Section 5.5, so we do not elaborate on further improvements to classical interior barrier methods.

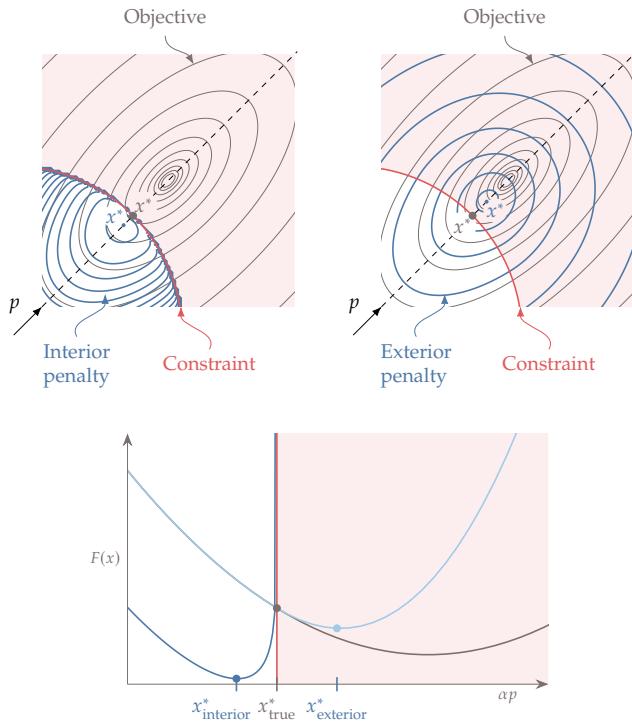
<sup>64</sup>. Murray, *Analytical expressions for the eigenvalues and eigenvectors of the Hessian matrices of barrier and penalty functions*. 1971

<sup>65</sup>. Forsgren et al., *Interior Methods for Nonlinear Optimization*. 2002

#### Example 5.14: Logarithmic penalty for inequality constrained problem

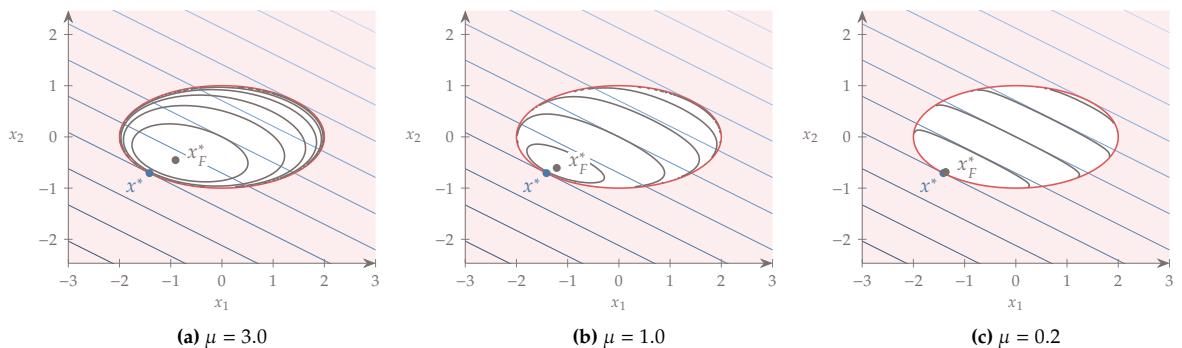
Consider the equality constrained problem from Ex. 5.5. The penalized function for that case using the logarithmic penalty (Eq. 5.56) is

$$F(x; \mu) = x_1 + 2x_2 - \mu \log\left(-\frac{1}{4}x_1^2 - x_2^2 + 1\right). \quad (5.57)$$



**Figure 5.21:** Interior penalties tend to infinity as the constraint is approached from the feasible side of the constraint (left), while exterior penalty functions activate when the points are not feasible (right). The minimum for both approaches is different from the true constrained minimum.

This function is shown in Fig. 5.22 for different values of the penalty parameter



$\mu$ . The penalized function is defined only in the feasible space, so we do not plot its contours outside the ellipse.

**Figure 5.22:** Logarithmic penalty for one inequality constraint. The minimum of the penalized function approaches the constrained minimum from the feasible side.

## 5.4 Sequential Quadratic Programming

Sequential quadratic programming (SQP) is the first of the modern constrained optimization methods we discuss. SQP is not a single algorithm, but instead, it is a conceptual method from which various specific algorithms have been derived. We begin with equality-constrained SQP, and then add inequality constraints.

### 5.4.1 Equality-Constrained SQP

To derive the SQP method, we start with the KKT conditions for this problem and treat them as residuals of equations that need to be solved. Recall that the Lagrangian is given by:

$$\mathcal{L}(x, \lambda) = f(x) + h(x)^T \lambda \quad (5.58)$$

The KKT conditions are that the derivatives of this function are zero:

$$\mathcal{R} = \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda) \\ \nabla_\lambda \mathcal{L}(x, \lambda) \end{bmatrix} = \begin{bmatrix} \nabla f(x) + [\nabla h(x)]^T \lambda \\ h(x) \end{bmatrix} = 0 \quad (5.59)$$

Recall that to solve a system of equations  $R(u) = 0$  using Newton's method, we solve the sequence of linear systems

$$[\nabla_u \mathcal{R}(u_k)] s_u = -\mathcal{R}(u_k), \quad (5.60)$$

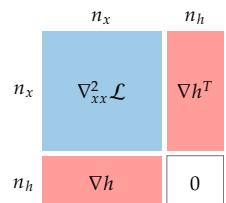
where  $s_u = u_{k+1} - u_k$ , and in our case  $u = [x^T, \lambda^T]^T$ . Differentiating the residual vector, Eq. 5.59, with respect to the two concatenated vectors in  $u$  yields the block linear system,

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L} & [\nabla h]^T \\ [\nabla h] & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix} \quad (5.61)$$

This is a linear system of  $n_x + n_h$  equations where the Jacobian matrix is square. The shape of the matrix and its subblocks are illustrated in Fig. 5.23. We solve a sequence of these problems to converge to the optimal design variables and the corresponding optimal Lagrange multipliers. At each iteration we update the design variables and Lagrange multipliers as:

$$x_{k+1} = x_k + s_x \quad (5.62)$$

$$\lambda_{k+1} = \lambda_k + s_\lambda \quad (5.63)$$



**Figure 5.23:** Shape of different submatrices forming the matrix for the SQP problem (Eq. 5.61)

SQP can be derived in an alternative way that leads to different insights. This alternate approach requires an understanding of quadratic programming (QP) discussed in more detail in Section 11.3, but briefly

described here. A QP problem is an optimization problem with a quadratic objective and linear constraints. In a general form, we can express any equality-constrained QP as:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Qx + f^T x \\ & \text{subject to} && Ax + b = 0 \end{aligned} \quad (5.64)$$

The objective is quadratic in  $x$  and the constraint is linear in  $x$ . A two-dimensional example is illustrated in Fig. 5.24. The constraint is a matrix equation that represents multiple linear equality constraints—one for every row in  $A$ .

We can solve this optimization problem analytically from the optimality conditions. First, we form the Lagrangian:

$$\mathcal{L}(x, \lambda) = \frac{1}{2}x^T Qx + f^T x + \lambda^T(Ax + b) \quad (5.65)$$

We now take the partial derivatives and set them equal to zero:

$$\begin{aligned} \nabla_x \mathcal{L} &= Qx + f + A^T \lambda = 0 \\ \nabla_\lambda \mathcal{L} &= Ax + b = 0 \end{aligned} \quad (5.66)$$

We can express those same equations in a block matrix form:

$$\begin{bmatrix} Q & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -f \\ -b \end{bmatrix} \quad (5.67)$$

This is like the procedure we used in solving the KKT conditions, except that these are just linear equations and so we can solve them directly without any iteration. Intuitively, it shouldn't be too surprising that finding the minimum of a quadratic objective (which means linear gradients) subject to linear constraints results in a set of linear equations.

As long as  $Q$  is positive definite, then the linear system always has a solution, and it is the global minimum of the QP<sup>†</sup>. The ease with which a QP can be solved provides an alternative motivation for SQP. For a general constrained problem we can make a local QP approximation of the nonlinear model, solve the QP, then repeat the process again. This method involves iteratively solving a sequence of quadratic programming problems, hence the name sequential quadratic programming.

To form the QP, we use a quadratic approximation of the Lagrangian (removing the constant term because it does not change the solution), and a linear approximation of the constraints, for some step,  $s$ , near our current point. In other words, we locally approximate the problem as

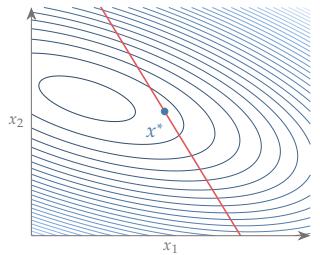


Figure 5.24: Quadratic problem in two dimensions.

<sup>†</sup>In other words, this is a *convex* problem. Convex optimization is discussed in Chapter 11.

the following QP:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}s^T \nabla_{xx}^2 \mathcal{L} s + \nabla_x \mathcal{L}^T s \\ & \text{by varying} && s \\ & \text{subject to} && [\nabla h]s + h = 0 \end{aligned} \quad (5.68)$$

We substitute the gradient of the Lagrangian into the objective:

$$\frac{1}{2}s^T \nabla_{xx}^2 \mathcal{L} s + \nabla f^T s + \lambda^T [\nabla h]s \quad (5.69)$$

Then, we substitute the constraint  $[\nabla h]s = -h$  into the objective:

$$\frac{1}{2}s^T \nabla_{xx}^2 \mathcal{L} s + \nabla f^T s - \lambda^T h \quad (5.70)$$

Now, we can remove the last term in the objective, because it does not depend on the design variables ( $s$ ) resulting in the following equivalent problem:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}s^T \nabla_{xx}^2 \mathcal{L} s + \nabla f^T s \\ & \text{by varying} && s \\ & \text{subject to} && [\nabla h]s + h = 0 \end{aligned} \quad (5.71)$$

Using the QP solution method outlined above, results in the following system of linear equations:

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} -\nabla f \\ -h \end{bmatrix} \quad (5.72)$$

We replace  $\lambda_{k+1} = \lambda_k + s_\lambda$  and multiply through:

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} + \begin{bmatrix} [\nabla h]^T \lambda_k \\ 0 \end{bmatrix} = \begin{bmatrix} -\nabla f \\ -h \end{bmatrix} \quad (5.73)$$

Subtracting the second term on both sides yields the same set of equations we found from applying Newton's method to the KKT conditions:

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix} \quad (5.74)$$

The derivation based on solving the KKT conditions is more fundamental. This alternative derivation relies the somewhat arbitrary choices (made in hindsight) of choosing a QP as the subproblem and using an approximation of the Lagrangian with constraints rather than an approximation of the objective with constraints, or an approximation of the Lagrangian with no constraints. Nevertheless, it is a useful conceptual model to consider the method as sequentially creating and solving QPs.

### 5.4.2 Inequality Constraints

If we have inequality constraints, we can still use the SQP algorithm for equality constrained problems introduced in the previous section, but we need some modifications as that approach assumes equality constraints. A common approach to this problem is to use an *active-set* method. If we knew which of the inequality constraints were active ( $g_i(x^*) = 0$ ) and which were inactive ( $g_i(x^*) < 0$ ) at the optimum, we could use the same solution approach, treating the active constraints as equality constraints and ignoring all inactive constraints. Unfortunately, we cannot assume any knowledge about which constraints are active at the optimum in general.

Finding which constraints are active in an iterative way poses a difficulty, as in general we would need to try all possible combinations of active constraints. This is intractable if there are many constraints.

While the true active set is not known until a solution is found, we can estimate it at each iteration. This estimated subset of active constraints is called the *working set*. Then, the linear system (5.61) can be solved only considering the equality constraints and the inequality constraints from the working set. The working set is then updated at each iteration.

As a first guess for the working set, we can simply evaluate the values for all inequality constraints, and select the ones for which  $g_k \geq -\epsilon$ , where  $\epsilon$  is a small positive quantity. This selects constraints that are near active or infeasible into the working set.

Many methods exist for updating the working set, only a general outline is discussed here. For the equality constrained case, determining the step direction,  $p_k$ , already ensures feasibility and so  $\alpha_k$  can be chosen without regarding the constraints. For the inequality constrained case, the computation of the Newton step,  $p_k$ , only involves the working set. Because this working set may be incomplete, the line search strategy needs to choose a step size that does not violate constraints outside the working set. The algorithm determines  $\alpha_{max}$ , which is the largest step size for which the constraints are still feasible. The line search then enforces  $\alpha_{max}$  as an upper bound. If  $\alpha_k < \alpha_{max}$  then the working set is unchanged. However, if  $\alpha_k = \alpha_{max}$  then the constraints that set the value for  $\alpha_{max}$  are said to be *blocking* (i.e., those constraints prevented the optimizer from taking a larger step). Those constraints are then added to the working set to improve the prediction of  $p_k$  for the next iteration.

---

**Tip 5.15:** How to handle maximum and minimum constraints.

Often a maximum or minimum constraint is desired. For example, the stress on a structure may be evaluated at many points and we want to make sure the maximum stress does not exceed a given yield stress:

$$\max(\sigma) \leq \sigma_y \quad (5.75)$$

However, the maximum function is not continuously differentiable. That is not always problematic, but it is also mathematically equivalent to constraining the stress at all  $m$  points, with the added benefit that all constraints are now continuously differentiable:

$$\sigma_i \leq \sigma_y \text{ for } i = 1 \dots m \quad (5.76)$$

While this adds many more constraints, if an active set method is used, there is little cost to adding more constraints as most of them will be inactive. Alternatively, a constraint aggregation method (Section 5.7) could be used.

---

### 5.4.3 Quasi-Newton SQP

The SQP method as discussed so far requires the Hessian of the Lagrangian  $\nabla_{xx}^2 \mathcal{L}$ , which if an exact Hessian is available, becomes Newton's method applied to the optimality condition. For the same reasons, and in a similar manner as discussed in Chapter 4, an exact Hessian is often not available, making it desirable to approximate the Hessian. We will denote the approximate Hessian of the Lagrangian, at iteration  $k$ , as  $W_k$ .

A high-level description of SQP with quasi-Newton approximations is provided in Alg. 5.16.<sup>†</sup> For the convergence criterion, we can use an infinity norm of the KKT system residual vector. To get more control over the convergence, we can have two separate tolerances for the norm of the optimality and feasibility.

<sup>†</sup>Sometimes linearizing the constraints can lead to an infeasible QP subproblem, and additional techniques are needed to treat these subproblems.<sup>56,66</sup>

56. Nocedal *et al.*, *Numerical Optimization*. 2006

66. Gill *et al.*, *SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization*. 2005

---

#### Algorithm 5.16: SQP with quasi-Newton approximation

##### Inputs:

$x_0$ : Starting point

$\tau_{\text{opt}}$ : Optimality tolerance

$\tau_{\text{feas}}$ : Feasibility tolerance

##### Outputs:

$x^*$ : Optimal point

$f(x^*)$ : Corresponding function value

---


$$\lambda_0 = 0$$

Initial Lagrange multipliers

$$W_0 = I$$

Initialize Hessian of Lagrangian approximation to identity matrix

$$k = 0$$

**while**  $\|\nabla_x \mathcal{L}\|_\infty > \tau_{\text{opt}}$  or  $\|h\|_\infty > \tau_{\text{feas}}$  **do**

Evaluate  $\nabla h_k, \nabla_x \mathcal{L}$

Solve KKT system (5.61) for  $s_x$  and  $s_\lambda$

$$\begin{bmatrix} W_k & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix}$$

Perform a line search in direction  $p_k = s_k$  using the merit function

$$\phi(\alpha, \mu) = f(x_k + \alpha p_k) + \mu \|h(x_k + \alpha p_k)\|$$

$$x_{k+1} = x_k + \alpha p_k$$

*Update step and active set*

$$\lambda_{k+1} = \lambda_k + s_\lambda$$

*Update the Lagrange multipliers*

$$\text{Update } W_{k+1}$$

*Compute quasi-Newton approximation using Eq. (5.77)*

Increment penalty parameter  $\mu$

$$k = k + 1$$

**end while**

---

Just as we did for unconstrained optimization, we can approximate  $W_k$  using the gradients of the Lagrangian and the BFGS update formula. However, unlike unconstrained optimization, we do not want the inverse of the Hessian directly. Instead, we make use of a version of the BFGS formula that computes the Hessian (rather than the inverse Hessian):

$$W_{k+1} = W_k - \frac{W_k s_k s_k^T W_k}{s_k^T W_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \quad (5.77)$$

where:

$$\begin{aligned} s_k &= x_{k+1} - x_k \\ y_k &= \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}). \end{aligned} \quad (5.78)$$

The step in the design variable space,  $s_k$ , is the step that resulted from the latest line search. The Lagrange multiplier is fixed to the latest value when approximating the curvature of the Lagrangian because we only need the curvature in the space of the design variables.

Recall that for the QP problem to have a solution then  $W_k$  must be positive definite. To ensure that this  $W_k$  is always positive definite, a *damped* BFGS update formula was devised.<sup>18§</sup> This method replaces the  $y$  with a new vector  $r$  defined as:

$$r_k = \theta_k y_k + (1 - \theta_k) W_k s_k, \quad (5.79)$$

where the scalar  $\theta_k$  is defined as

$$\theta_k = \begin{cases} 1 & \text{if } s_k^T y_k \geq 0.2 s_k^T W_k s_k, \\ \frac{0.8 s_k^T W_k s_k}{s_k^T W_k s_k - s_k^T y_k} & \text{if } s_k^T y_k < 0.2 s_k^T W_k s_k, \end{cases} \quad (5.80)$$

18. Powell, *Algorithms for nonlinear constraints that use Lagrangian functions*. 1978

§The damped BFGS formula is not always the best approach for all problems, and other Lagrangian Hessian approximation methods exist like the symmetric rank-one approximation.<sup>61</sup> Additionally, for very large problems storing a dense Hessian may be problematic and so limited-memory update methods are used.<sup>67</sup>

61. Fletcher, *Practical Methods of Optimization*. 1987

67. Liu et al., *On the limited memory BFGS method for large scale optimization*. 1989

which can range from 0 to 1. We then use the same BFGS update formula, Eq. 5.77, except that we replace each  $y_k$  with  $r_k$ .

To better understand what this method is doing, take a closer look at the two extremes for  $\theta$ . If  $\theta_k = 0$  then Eq. 5.79 in combination with Eq. 5.77 yields  $W_{k+1} = W_k$ ; that is, the Hessian approximation is unmodified. At the other extreme,  $\theta_k = 1$  yields the full BFGS update formula ( $r_k$  is set to  $y_k$ ). Thus, the parameter  $\theta_k$  provides a linear weighting between keeping the current Hessian approximation and a full BFGS update.

The definition of  $\theta_k$  (5.80) ensures that  $W_{k+1}$  stays close enough to  $W_k$  and remains positive definite. The damping is activated when the predicted curvature in the new latest step is below one fifth of the curvature predicted by the latest approximate Hessian. This could happen when the function is flattening or when the curvature becomes negative.

Like the quasi-Newton methods we discussed in unconstrained optimization, we solve the QP for a search direction,  $p_x$  (as opposed to a full step  $s_x$ ), and perform a line search. However, we cannot just use the objective function as the metric of our line search as we did in unconstrained optimization; we need to use some kind of merit function (a form of penalty) or filter. These are techniques used to evaluate whether a step is acceptable in a line search. The details of these techniques are discussed later in Section 5.6, since they are used for both SQP and interior-point methods.

---

**Example 5.17:** SQP applied to equality-constrained problem

We now solve Ex. 5.4 using the SQP method (Alg. 5.16). The gradient of the equality constraint is

$$\nabla h = \begin{bmatrix} \frac{1}{2}x_1 \\ 2x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$

and differentiating the Lagrangian with respect to  $x$  yields

$$\frac{\partial \mathcal{L}}{\partial x} = \begin{bmatrix} 1 + \frac{1}{2}\lambda x_1 \\ 2 + 2\lambda x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

We start at  $x^{(0)} = [2, 1]$  with an initial Lagrange multiplier  $\lambda = 0$ . We set the initial estimate of the Lagrangian Hessian to  $W_0 = I$ . The KKT system (5.61) to be solved is then

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} s_{x_1} \\ s_{x_2} \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \\ -1 \end{bmatrix}.$$

The solution of  $s$  to the above is  $[-0.2, -0.4, -0.8]$ . Performing a line search in the direction  $p = [-0.2, -0.4]$  yields the solution to the next iteration at  $x^{(1)} = [1.8, 0.6]$ . The Lagrange multiplier is updated to  $\lambda_1 = -0.8$ .

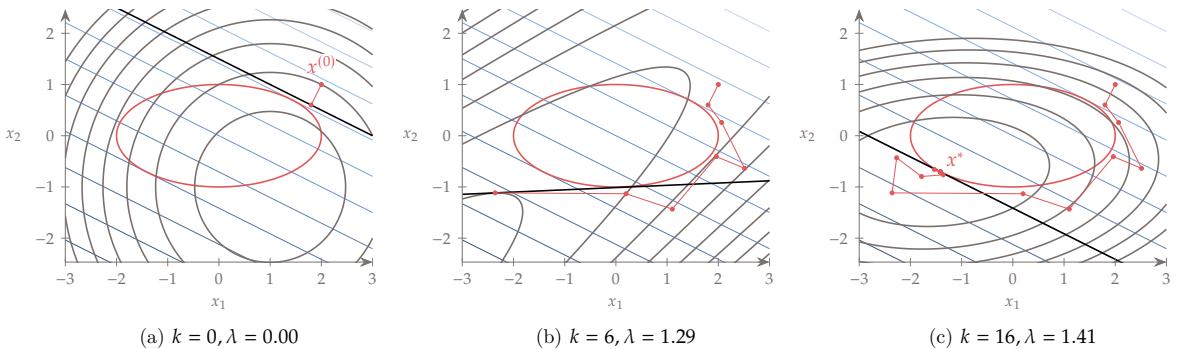
To update the approximate Hessian  $W_k$  using the damped BFGS update formula (5.79), we need to compare the values of  $s_0^T y_0 = -0.272$  and  $s_0^T W_0 s_0 = 0.2$ . Since  $s_k^T y_k < 0.2 s_k^T W_k s_k$ , we need to compute the scalar  $\theta = 0.339$  using Eq. 5.80. This provides a partial BFGS update where  $r_0$  is a mix of the current Hessian approximation and the step  $y_0$ . Using the quasi-Newton update Eq. 5.77, we get the approximate Hessian for the next iteration as

$$W_1 = \begin{bmatrix} 1.076 & -0.275 \\ -0.275 & 0.256 \end{bmatrix}.$$

The penalty parameter  $\mu$  also needs to be incremented. We use a factor of 1.1 in this example. This process is then repeated for subsequent iterations.

Fig. 5.25 shows SQP optimization at various iterations. The gray contour is  $\frac{1}{2}s^T W s + \nabla f^T s$ , the QP subproblem (5.71) that is being solved at each iteration, and  $W$  is the approximate Hessian updated using the quasi-Newton method. The linearized constraint  $[\nabla h]s + h = 0$  is also shown as a straight line.

The starting point is infeasible and the iterations remain infeasible until the last few iterations. This behavior is common for SQP, because while it satisfies



the linear approximation of the constraints at each step, it does not necessarily satisfy the nonlinear constraint of the actual problem. As the approximation of the constraint becomes more accurate near the solution, the nonlinear constraint is then satisfied.

**Figure 5.25:** SQP algorithm iterations.

## 5.5 Interior Point Methods

One way to look at interior point methods is to make a seemingly small (but important!) modification from the interior penalty method we have

already seen. We formulate the constrained optimization problem as:

$$\begin{aligned}
 & \text{minimize} && f(x) - \mu \sum_k \ln s_k \\
 & \text{by varying} && x, s \\
 & \text{subject to} && h_j(x) = 0 \text{ for } j = 1, \dots, n_h \\
 & && g_k(x) + s_k = 0 \text{ for } k = 1, \dots, n_g
 \end{aligned} \tag{5.81}$$

One difference relative to the barrier method is that rather than treating the problem as unconstrained, we apply Newton's method to the KKT conditions like we do in SQP methods. Implicit in the constraints is  $s_k \geq 0$ . This condition is enforced by the logarithm function which prevents  $s$  from approaching zero. Because  $s_k$  is always positive that means that at the solution  $g_k(x^*) < 0$ , which satisfies the inequality constraints. Like the penalty method, this formulation is not actually equivalent to our original constrained problem, except in the limit as  $\mu \rightarrow 0$ . We will need to solve a sequence of solutions to this problem with  $\mu$  approaching zero. Unlike SQP, this formulation uses only equality constraints; and so it avoids the combinatorial problem of trying to determine an active set.

First, we form the Lagrangian for this problem as

$$\mathcal{L}(x, \lambda, \sigma) = f(x) - \mu \sum_k \ln s_k + h(x)^T \lambda + (g(x) + s)^T \sigma. \tag{5.82}$$

By taking derivatives with respect to  $x$ ,  $\lambda$ ,  $\sigma$ , and  $s$  respectively, the KKT conditions for this problem are:

$$\begin{aligned}
 \frac{\partial f}{\partial x_i} + \lambda_j \frac{\partial h_j}{\partial x_i} + \sigma_k \frac{\partial g_k}{\partial x_i} &= 0 \\
 h_j &= 0 \\
 g_k + s_k &= 0 \\
 -\mu \sum_k \frac{1}{s_k} + \sigma_k &= 0
 \end{aligned} \tag{5.83}$$

We can also express this in vector form by defining a matrix  $S$ , which is a diagonal matrix with  $S_{kk} = s_k$ , and defining a vector of ones as  $\mathbf{1}$ .

$$\begin{aligned}
 \nabla f(x) + [\nabla h(x)]^T \lambda + [\nabla g(x)]^T \sigma &= 0 \\
 h &= 0 \\
 g + s &= 0 \\
 -\mu S^{-1} \mathbf{1} + \sigma &= 0
 \end{aligned} \tag{5.84}$$

This form of the equation can pose numerically difficulties as  $s \rightarrow 0$  (perhaps most obvious by examining the last equation in Eq. 5.83), and so we will multiply the last equation through by  $S$ .

$$\begin{aligned} \nabla f(x) + [\nabla h(x)]^T \lambda + [\nabla g(x)]^T \sigma &= 0 \\ h &= 0 \\ g + s &= 0 \\ -\mu \mathbf{1} + S\sigma &= 0 \end{aligned} \tag{5.85}$$

We now have a set of residual equations and can apply Newton's method, just like we did for SQP. The result is:

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L}(x) & [\nabla h(x)]^T & [\nabla g(x)]^T & 0 \\ \nabla h(x) & 0 & 0 & 0 \\ \nabla g(x) & 0 & 0 & I \\ 0 & 0 & S & \Sigma \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \\ s_\sigma \\ s_s \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda, \sigma) \\ h(x) \\ g(x) + s \\ S\sigma - \mu \mathbf{1} \end{bmatrix} \tag{5.86}$$

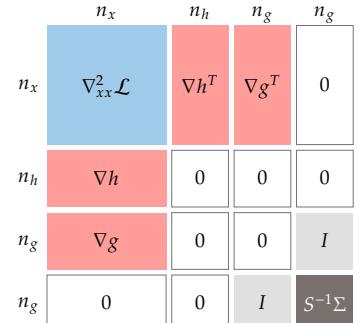
where  $\Sigma$  is a diagonal matrix with  $\sigma$  across the diagonal, and  $I$  is the identity matrix.

For numerical efficiency, we make some small modifications to this system. The matrix is almost symmetric and with a little work we can make it symmetric. If we multiply the last equation by  $S^{-1}$  we have:

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L}(x) & [\nabla h(x)]^T & [\nabla g(x)]^T & 0 \\ \nabla h(x) & 0 & 0 & 0 \\ \nabla g(x) & 0 & 0 & I \\ 0 & 0 & I & S^{-1}\Sigma \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \\ s_\sigma \\ s_s \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda, \sigma) \\ h(x) \\ g(x) + s \\ \sigma - \mu S^{-1} \mathbf{1} \end{bmatrix} \tag{5.87}$$

The advantage of this equivalent system is that we can use a symmetric linear system solver, which is more efficient than that of a general linear system solver. The shape of the matrix and its blocks are illustrated in Fig. 5.26.

The steps in the interior point method are detailed in Alg. 5.18. Similarly to quasi-Newton SQP, we estimate of the Hessian of the Lagrangian and performs a line search, but the system of equations that we solve is different. Another major difference is that there is no active set to update. ¶



**Figure 5.26:** Shape of the blocks in the interior point system (5.87)

¶Many important implementation details and variations exist on things like: how often and in what manner to update the penalty parameter  $\mu$ , ensuring that the BFGS Hessian is positive definite and is computed in a memory-efficient manner for large problems, resetting parameters, etc. Further implementation details can be found in the literature.<sup>68,69</sup>

68. Wächter et al., *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*. 2005

69. Byrd et al., *An Interior Point Algorithm for Large-Scale Nonlinear Programming*. 1999

---

**Algorithm 5.18:** Interior point method with quasi-Newton approximation

**Inputs:**

$x_0$ : Starting point

$\tau_{\text{opt}}$ : Optimality tolerance

$\tau_{\text{feas}}$ : Feasibility tolerance

**Outputs:** $x^*$ : Optimal point $f(x^*)$ : Corresponding function value

$$\begin{array}{ll} \lambda_0 = 0; \sigma_0 = 0 & \text{Initial Lagrange multipliers} \\ s_0 = 1 & \text{Initial slack variables} \\ W_0 = I & \text{Initialize Hessian of Lagrangian approximation to identity matrix} \\ k = 0 & \\ \end{array}$$

**while**  $\|\nabla_x \mathcal{L}\|_\infty > \tau_{\text{opt}}$  or  $\|h\|_\infty > \tau_{\text{feas}}$  **do**Evaluate  $\nabla h_k, \nabla_x \mathcal{L}$ Solve the KKT system (5.87) for  $s$ 

$$\begin{bmatrix} W_k & [\nabla h(x)]^T & [\nabla g(x)]^T & 0 \\ \nabla h(x) & 0 & 0 & 0 \\ \nabla g(x) & 0 & 0 & I \\ 0 & 0 & I & S^{-1}\Sigma \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \\ s_\sigma \\ s_s \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda, \sigma) \\ h(x) \\ g(x) + s \\ \sigma - \mu S^{-1}\mathbf{1} \end{bmatrix}$$

Perform a line search in direction  $p_k = s_k$  using the merit function

$$\phi(\alpha, \mu) = f(x_k + \alpha p_k) + \mu \|h(x_k + \alpha p_k)\| + \max(0, g(x_k + \alpha p_k))$$

$$\begin{array}{ll} x_{k+1} = x_k + \alpha p_k & \text{Update step} \\ \lambda_{k+1} = \lambda_k + s_\lambda & \text{Update the Lagrange multipliers} \\ \text{Update } W_{k+1} & \text{Compute quasi-Newton approximation using Eq. (5.77)} \\ \text{Reduce penalty parameter } \mu & \\ k = k + 1 & \\ \end{array}$$

**end while**

Generally speaking, active-set SQP methods are more effective on medium-scale problems, while interior-point methods are more effective on large-scale problems. Interior-point methods are also generally more sensitive to the initial starting point and the scaling of the problem.<sup>56</sup> These are of course only generalities, and are not replacements for testing multiple algorithms on the problem of interest. Many optimization frameworks make it easy to switch between optimization algorithms facilitating this type of testing.

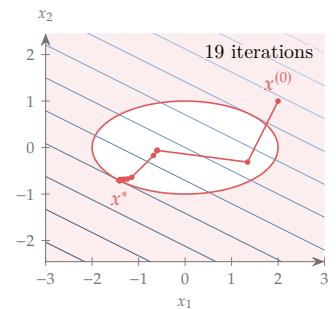
56. Nocedal et al., Numerical Optimization. 2006

**Example 5.19:** Interior point method applied to inequality-constrained problem

Here we solve Ex. 5.5 using the interior point method (Alg. 5.18) starting at  $x^{(0)} = [2, 1]$ . The initial Lagrange multiplier is  $\sigma = 0$  and the slack variable is  $s = 1$ . Starting with a penalty parameter of  $\mu = 20$  results in the iterations shown in Fig. 5.27.

For the first iteration, differentiating the Lagrangian with respect to  $x$  yields

$$\frac{\partial \mathcal{L}}{\partial x} = \begin{bmatrix} 1 + \frac{1}{2}\sigma x_1 \\ 2 + 2\sigma x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$



**Figure 5.27:** Interior point algorithm iterations.

and the gradient of the constraint is

$$\nabla g = \begin{bmatrix} \frac{1}{2}x_1 \\ 2x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

The interior point system of equations (5.87) at the starting point is

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s_{x_1} \\ s_{x_2} \\ s_\sigma \\ s_s \end{bmatrix} = \begin{bmatrix} -1 \\ -2 \\ -2 \\ 20 \end{bmatrix}.$$

The solution is  $s = [-21, -42, 20, 103]$ . Performing a line search in the direction  $p = [-21, -42]$  yields  $x^{(1)} = [1.34375, -0.3125]$ . The Lagrange multiplier and slack variable are updated to  $\sigma_1 = 20$  and  $s_1 = 104$ , respectively.

To update the approximate Hessian  $W_k$ , we can use the damped BFGS update formula (5.79) to ensure that  $W_k$  is positive definite. By comparing  $s_0^T y_0 = 73.21$  and  $s_0^T W_0 s_0 = 2.15$ , we can see that  $s_k^T y_k \geq 0.2s_k^T W_k s_k$ , and therefore we do a full BFGS update with  $\theta_0 = 1$  and  $r_0 = y_0$ . Using the quasi-Newton update Eq. 5.77, we get the approximate Hessian

$$W_1 = \begin{bmatrix} 1.388 & 4.306 \\ 4.306 & 37.847 \end{bmatrix}.$$

We reduce the penalty parameter  $\mu$  by a factor of 2 at each iteration. This process can be repeated for subsequent iterations.

The starting point is infeasible but the algorithm finds a feasible point after the first iteration. From then on, it approaches the optimum from within the feasible region, which is the expected behavior for interior point algorithms, as shown in Fig. 5.27.

**Tip 5.20:** Some equality constraints can be posed as inequality constraints.

Equality constraints are less common in engineering design problems than inequality constraints. Sometimes we pose a problem as an equality constraint unnecessarily. For example, the simulation of an aircraft in steady-level flight may want the lift to equal the weight. Formally, this is an equality constraint, but it can also be posed as an inequality constraint of the form:  $L \geq W$ . This is because there is no advantage to additional lift, as it will increase drag, and so the constraint will always be active at the solution. While an equality constraint is perhaps more natural algorithmically, the flexibility of the inequality constraint can often allow the optimizer to explore the design space more effectively. Consider another example: a propeller may be designed to match a specified thrust target. While an equality constraint would likely work, it is more constraining than necessary. If the optimal design was somehow able to produce excess thrust we would not reject that design. Thus, we shouldn't formulate the constraint in a way that is unnecessarily restrictive.

---

## 5.6 Merit Functions and Filters

For unconstrained optimization, evaluating the effectiveness of the line search is relatively straightforward. The primary concern is that the objective function achieves a sufficient decrease. For constrained optimization, the problem is not as simplistic. A new point may decrease the objective but increase in infeasibility. It is not obvious how best to weigh these tradeoffs. We need to be able to combine these metrics into one metric for the purposes of evaluating the line search. We cannot just use the Lagrangian because it is not computed at each point in the line search. Traditionally, this issue has been dealt with by using merit functions.

Common merit functions have already been introduced in the form of penalty functions. These include:  $l_1$  and  $l_2$  norms of constraint violations:

$$F(x; \mu) = f(x) + \mu \|c_{vio}(x)\|_p \text{ for } p = 1 \text{ or } 2, \quad (5.88)$$

and the augmented Lagrangian:

$$F(x; \mu) = f(x) + \lambda(x)^T h(x) + \frac{1}{2} \mu \|c_{vio}(x)\|_2^2. \quad (5.89)$$

where  $c_{vio}$  are the constraint violations defined as:

$$c_{vio,i}(x) = \begin{cases} |h_i(x)| & \text{for equality constraints} \\ \max(0, g_i(x)) & \text{for inequality constraints} \end{cases} \quad (5.90)$$

One downside to merit functions, similar to that seen with penalty functions, is that it is in general difficult to choose a suitable value for the penalty parameter. If needs to be large to ensure that there is improvement, but if it is too large then often a full Newton step is not permitted and convergence is slowed.

A more recent approach is a filter method.<sup>70</sup> Filter methods, in general, provide less interference with a full Newton step<sup>71</sup>, and are effective for both sequential quadratic programming and interior point methods.<sup>72</sup> The approach is based on concepts from multiobjective optimization, which is discussed in more detail in Chapter 9 (Fig. 9.1 in particular). The basic idea is that we accept a point in the line search if it is not *dominated* by points in the current filter. One point dominates another if its objective is lower *and* the sum of its constraint violations is lower. If a point is acceptable to the line search it is added to the filter,

<sup>70</sup>. Fletcher et al., *Nonlinear programming without a penalty function*. 2002

<sup>71</sup>. Fletcher et al., *A Brief History of Filter Methods*. 2006

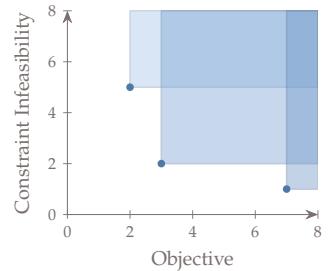
<sup>72</sup>. Benson et al., *Interior-Point Methods for Nonconvex Nonlinear Programming: Filter Methods and Merit Functions*. 2002

and any dominated points are removed from the filter (as they are no longer necessary).

**Example 5.21:** Using a filter.

A filter consists of pairs  $(f(x), c(x))$  where  $c(x)$  is the sum of constraint violations:  $c(x) = \|c_{vio}(x)\|_1$ . As an example, assume that the current filter contains these three points:  $(2, 5)$ ,  $(3, 2)$ , and  $(7, 1)$ . Notice that none of the points in the filter dominates any other. We could plot these points as shown in Fig. 5.28, where the shaded region correspond to areas that are dominated by the points in the filter. During a line search a new candidate point is evaluated. There are three possible outcomes. Let us consider three example points that illustrate these three outcomes.

1.  $(1, 4)$ : this point is not dominated by any point in the filter. The step is accepted, and this point is added to the filter. Additionally it dominates one of the points in the filter,  $(2, 5)$ , and so that point is removed from the filter. The current filter is now  $(1, 4)$ ,  $(3, 2)$ , and  $(7, 1)$ .
2.  $(1, 6)$ : this point is not dominated by any point in the filter. The step is accepted, and this new point is added to the filter. No points in the filter are dominated and so nothing is removed. The current filter is now  $(1, 6)$ ,  $(2, 5)$ ,  $(3, 2)$ , and  $(7, 1)$ .
3.  $(4, 3)$ : this point is dominated by a point in the filter  $(3, 2)$ . The step is rejected and the line search must continue. The filter is unchanged.



**Figure 5.28:** A filter method example showing three points in the filter, the shaded regions correspond to points that are dominated by the filter.

The outline of this section presents only the basic ideas. Robust implementation of a filter method requires imposing *sufficient* decrease conditions, not unlike those in the unconstrained case, as well as a few other minor modifications.<sup>71</sup>

<sup>71</sup>Fletcher et al., *A Brief History of Filter Methods*. 2006

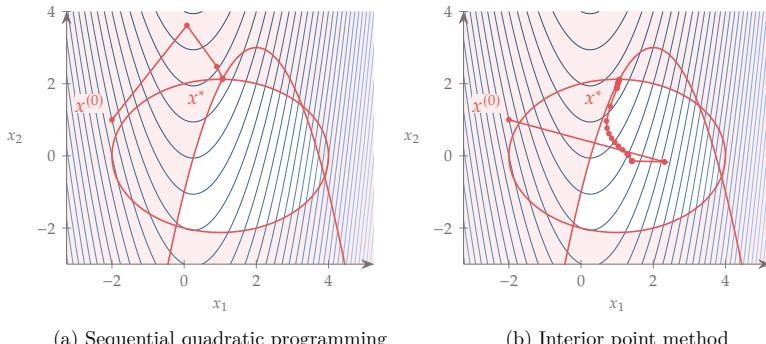
**Tip 5.22:** Consider reformulating your constraints.

There are often multiple mathematically equivalent ways to pose the problem constraints. Sometimes reformulating can yield equivalent problems that are significantly easier to solve. In some cases it can help to add constraints that are redundant, but guide the optimizer to more useful areas of the design space. Similarly, one should consider whether residuals should be solved internally or posed as constraints at the optimizer level.

**Example 5.23:** Numerical solution of graphical solution example.

Recall the constrained problem with a quadratic objective and quadratic constraints introduced in Ex. 5.1. Instead of finding an approximate solution

graphically or trying to solve this analytically, we can now solve this numerically using SQP or the interior point method. The resulting optimization paths are shown in Fig. 5.29. The difference in the number of iterations between these two methods is not representative because this is a simple problem and their relative performance is highly dependent on implementation details.

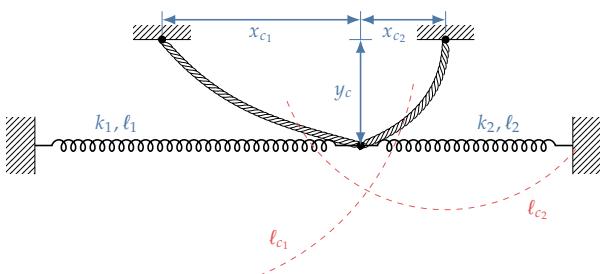


**Figure 5.29:** Numerical solution of Ex. 5.1.

---

**Example 5.24:** Constrained spring system.

Consider the spring system from Ex. 4.22, which is an unconstrained optimization problem. We can constrain the spring system by attaching two cables as shown in Fig. 5.30, where  $\ell_{c1} = 9$  m,  $\ell_{c2} = 6$  m,  $y_c = 2$  m,  $x_{c1} = 7$  m, and  $x_{c2} = 3$  m. Because the cables do not resist compression forces, they

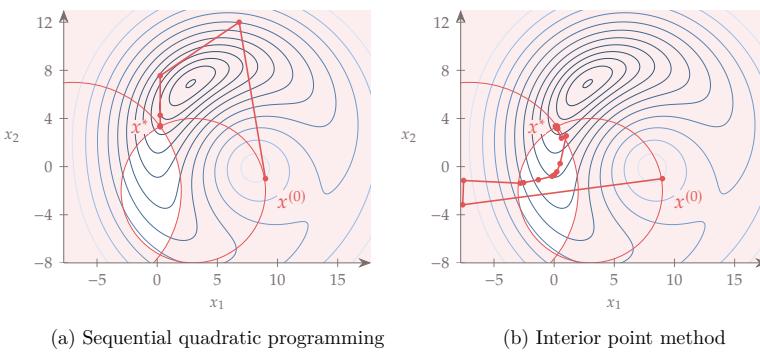


**Figure 5.30:** Spring system constrained by two cables.

correspond to inequality constraints, which yields the following problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} k_1 \left( \sqrt{(\ell_1 + x_1)^2 + x_2^2} - \ell_1 \right)^2 + \frac{1}{2} k_2 \left( \sqrt{(\ell_2 - x_1)^2 + x_2^2} - \ell_2 \right)^2 - mgx_2 \\ \text{by varying} \quad & x_1, x_2 \\ \text{subject to} \quad & \sqrt{(x_1 + x_{c1})^2 + (x_2 + y_c)^2} \leq \ell_{c1}, \\ & \sqrt{(x_1 - x_{c2})^2 + (x_2 + y_c)^2} \leq \ell_{c2}. \end{aligned} \quad (5.91)$$

The optimization paths for SQP and interior point method are shown in Fig. 5.31.



**Figure 5.31:** Optimization of constrained spring system.

## 5.7 Constraint Aggregation

As discussed in Chapter 6, adjoint methods, or reverse mode AD, are effective for scenarios with many inputs and few outputs. These methods are desirable because they allow us to work with large numbers of design variables. However, they are only effective if we have a small number of constraints. In many practical engineering problems we have many constraints, and so in these scenarios we can use *constraint aggregation* methods. A constraint aggregation function combines some or all of the constraints into a single constraint such that violation of any single constraint causes the total function to be violated (approximately). A simple example would be the max function. If  $\max(g(x)) < 0$  then we know that all of  $g_j(x) < 0$ . However, the max function is not differentiable and so alternative functions that play a similar role are needed.

A common constraint aggregation function is the KS function<sup>73</sup>,

<sup>73</sup> Kreisselmeier et al., *Systematic Control Design by Optimizing a Vector Performance Index*. 1979

which acts like a soft maximum:

$$KS(x) = \frac{1}{\rho} \ln \left( \sum_{j=1}^m e^{\rho g_j(x)} \right) \quad (5.92)$$

where  $\rho$  is an aggregation parameter, like a penalty parameter used in penalty methods.

**Example 5.25:** Constrained spring system with aggregated constraints.

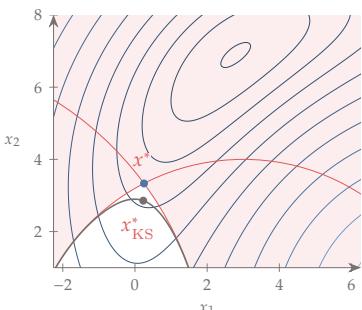
Consider the constrained spring system from Ex. 5.24. Aggregating the two constraints using the KS function, we can formulate a single constraint as

$$KS(x_1, x_2) = \frac{1}{\rho} \ln \left( e^{\rho g_1(x_1, x_2)} + e^{\rho g_2(x_1, x_2)} \right), \quad (5.93)$$

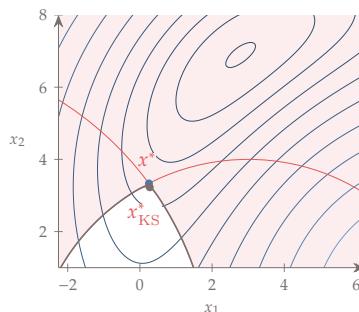
where

$$\begin{aligned} g_1(x_1, x_2) &= \sqrt{(x_1 + x_{c1})^2 + (x_2 + y_c)^2} - \ell_{c1}, \\ g_2(x_1, x_2) &= \sqrt{(x_1 - x_{c2})^2 + (x_2 + y_c)^2} - \ell_{c2}. \end{aligned} \quad (5.94)$$

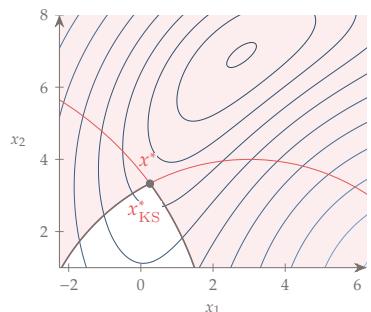
We plot the contour of  $KS = 0$  in Fig. 5.32 for increasing values of the aggregation parameter  $\rho$ . We can see the difference in the feasible region for the lowest value of  $\rho$ , which results in a conservative optimum. For the highest value of  $\rho$ , the optimum obtained with constraint aggregation is graphically indistinguishable, and the objective function value approaches the true optimal value of -22.1358.



(a)  $\rho_{KS} = 2, f_{KS}^* = -19.448$



(b)  $\rho_{KS} = 10, f_{KS}^* = -21.653$



(c)  $\rho_{KS} = 100, f_{KS}^* = -22.090$

**Figure 5.32:** Spring system constrained by two cables.

## 5.8 Summary

With constraints, we can no longer use  $\nabla f$  as a first-order convergence criterium, but rather define a new function called the Lagrangian

and use  $\nabla \mathcal{L}$  as our first-order convergence criteria. The Lagrangian is a function not just of the design variables, but also the Lagrange multipliers and slack variables. The enumeration of that first-order convergence criteria,  $\nabla \mathcal{L}$  (along with a constraint on the sign of the Lagrange multipliers associated with the inequality constraints), is called the KKT conditions.

Penalty methods are useful as a beginning conceptual model, and for use in gradient-free methods, but are no longer used for constrained gradient-based optimization. Instead, sequential quadratic programming and interior point methods are the state of the art. These methods are applications of Newton's method to the KKT conditions. One primary difference is in the treatment of inequality constraints. Sequential quadratic programming methods try to distinguish between active and inactive constraints, using the (potentially) active constraints like equality constraints and ignoring the (potentially) inactive ones. Interior point methods add slack variables to force all constraints to behave like equality constraints.

## Problems

5.1 Answer *true* or *false* and correct the false statements.

- a) Penalty methods are among the most effective methods for constrained optimization.
- b) For an equality constraint in  $n$ -dimensional space, all feasible directions about a point are perpendicular to the constraint gradient at that point and define a plane with  $n - 1$  degrees of freedom.
- c) The feasible directions about a point on an inequality constraint define a half space whose dividing hyperplane is perpendicular by the gradient of the constraint at that point.
- d) A point is optimal if there is only one feasible direction that is also a descent direction.
- e) For an inequality-constrained problem, if we replace the inequalities that are active at the optimum with equality constraints and ignore the inactive constraints, we get the same optimum.
- f) For a point to be optimal, the Lagrangian multipliers for both equality and active inequality constraints must be positive.
- g) The complementarity conditions are easy to solve for because either the Lagrange multiplier is zero or the slack variable is zero.

- h) At the optimum of a constrained problem, the Hessian of the objective function must be positive semidefinite.
- i) The Lagrange multipliers represent the change in the objective function we would get for a perturbation in the constraint value.
- j) Sequential quadratic programming seeks to find the solution of the KKT system.
- k) Interior point methods must start with a point in the interior of the feasible region.
- l) Constraint aggregation combines multiple constraints into a single constraint that is equivalent.

5.2 Consider Ex. 5.4 and modify it so that the equality constraint is the negative of the original one, that is,

$$h(x_1, x_2) = -\frac{1}{4}x_1^2 - x_2^2 + 1 = 0.$$

Classify the critical points and compare them with the original solution. What does that tell you about the significance of the Lagrange multiplier sign?

5.3 Similarly to the previous exercise, consider Ex. 5.5 and modify it so that the inequality constraint is the negative of the original one, that is,

$$h(x_1, x_2) = -\frac{1}{4}x_1^2 - x_2^2 + 1 \leq 0.$$

Classify the critical points and compare them with the original solution.

5.4 Consider the following optimization problem,

$$\begin{aligned} & \text{minimize} && x_1^2 + 3x_2^2 + 4 \\ & \text{by varying} && x_1, x_2 \\ & \text{subject to} && x_2 \geq 1 \\ & && x_1^2 + 4x_2^2 \leq 4 \end{aligned} \tag{5.95}$$

Find the optimum analytically.

5.5 Find the rectangle of maximum area that can be inscribed in an ellipse. Give your answer in terms of the ratio of the two areas. Check that your answer is intuitively correct for the special case of a rectangle inscribed in a circle.

5.6 In Section 2.1, we mentioned that Euclid showed that among rectangles of a given perimeter, the square has the largest area. Formulate the problem and confirm the result analytically. What are the units and the physical interpretation of the Lagrange multiplier in this problem? *Exploration:* Show that if you minimize the perimeter with an area constrained to the optimum value you found above, you get the same solution.

5.7 *Column in Compression.* Consider a thin-walled tubular column subjected to a compression force. We want to minimize the mass of the column while ensuring that the structure does not yield or buckle under a compression force of magnitude  $F$ . Our design variables are the radius of the tube ( $R$ ) and the wall thickness ( $t$ ). This design optimization problem can be stated as,

$$\begin{aligned} & \text{minimize} && 2\rho\ell\pi R t && \text{mass} \\ & \text{by varying} && R, t && \text{radius, wall thickness} \\ & \text{subject to} && \sigma_{\text{yield}} - \frac{F}{2\pi R t} \leq 0 && \text{yield stress} \\ & && F - \frac{\pi^3 E R^3 t}{4\ell^2} \leq 0 && \text{buckling load} \end{aligned}$$

In the formula for the mass in the objective above,  $\rho$  is the material density, and we assume that  $t \ll R$ . The first constraint is the compressive stress, which is simply the force divided by the cross-sectional area. The second constraint uses Euler's critical buckling load formula, where  $E$  is the material Young's modulus, and the second moment of area is replaced with the one corresponding to a circular cross section ( $I = \pi R^3 t$ ).

Find the optimum  $R$  and  $t$  as a function of the other parameters. Pick some reasonable values for the parameters and verify your solution graphically. Plot the gradients of the objective and constraints at the optimum and verify the Lagrange multipliers graphically.

5.8 *Beam with H section.* Consider a cantilevered beam with an H-shape cross-section composed of a web and flanges subject to a transverse load as shown in Fig. 5.34. The objective is to minimize the structural weight by varying the web thickness  $t_w$  and the flange thickness  $t_b$ , subject to stress constraints. The other cross-sectional parameters are fixed; the web height  $h$  is 250 mm and the flange width  $b$  is 125 mm. The axial stress in the flange and the shear stress in the web should not exceed the corresponding

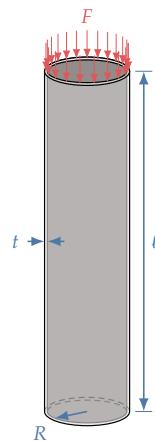


Figure 5.33: Slender tubular column in compression

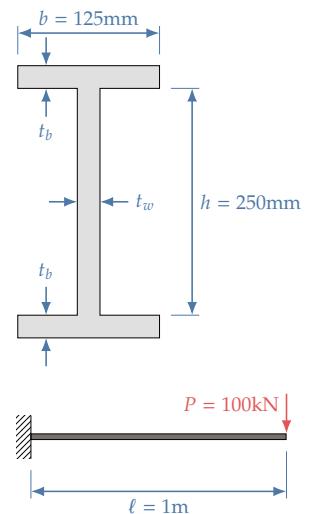


Figure 5.34: Cantilever beam with H section.

yield values ( $\sigma_{\text{yield}} = 200 \text{ MPa}$  and  $\tau_{\text{yield}} = 116 \text{ MPa}$ , respectively). The optimization problem can be stated as,

$$\begin{array}{lll} \text{minimize} & 2bt_b + ht_w & \text{mass} \\ \text{by varying} & t_b, t_w & \text{flange and web thicknesses} \\ \text{subject to} & \frac{P\ell h}{2I} - \sigma_{\text{yield}} \leq 0 & \text{axial stress} \\ & \frac{1.5P}{ht_w} - \tau_{\text{yield}} \leq 0 & \text{shear stress} \end{array}$$

The second moment of area is

$$I = \frac{h^3}{12}t_w + \frac{b}{6}t_b^3 + \frac{h^2b}{2}t_b.$$

Find the optimal values of  $t_b$  and  $t_w$  by solving the KKT conditions analytically. Plot the objective contours and constraints to verify your result graphically.

5.9 *Penalty method implementation.* Program one or more penalty methods from Section 5.3.

- a) Solve the constrained problem from Ex. 5.8 as a first test of your implementation. Use an existing software package for the optimization subproblem or the unconstrained optimizer you implemented in Prob. 4.9. How far can you push the penalty parameter until the optimizer fails? How close can you get to the exact optimum? Try different starting points and verify that the algorithms always converges to the same optimum.
- b) Solve the problem from Prob. 5.3.
- c) Solve the problem detailed in Prob. 5.11.
- d) *Exploration:* Solve any other problem from this section or a problem of your choosing.

5.10 *Constrained optimizer implementation.* Program a SQP or interior point algorithm. You may repurpose the BFGS algorithm that you implemented in Prob. 4.9. For SQP, start by implementing only equality constraints and reformulate test problems with inequality constraints as problems with only equality constraints.

- a) Reproduce the results from Ex. 5.17 (SQP) or Ex. 5.19 (interior point).
- b) Solve the problem from Prob. 5.3.

- c) Solve the problem detailed in Prob. 5.11.
- d) Compare the computational cost, precision, and robustness of your optimizer with an existing software package.

5.11 *Aircraft Fuel Tank.* A jet aircraft needs to carry a streamlined external fuel tank with a required volume. The tank shape is approximated as an ellipsoid. We want to minimize the drag of the fuel tank by varying its length and diameter, that is,

$$\begin{aligned} & \text{minimize} && D(\ell, d) \\ & \text{by varying} && \ell, d \\ & \text{subject to} && V_{\text{req}} - V(\ell, d) \leq 0. \end{aligned}$$

The drag is given by,

$$D = \frac{1}{2} \rho v^2 C_D S,$$

where the air density is  $\rho = 0.55 \text{ kg/m}^3$ , the aircraft speed is  $v = 300 \text{ m/s}$ . The drag coefficient of an ellipsoid can be estimated as<sup>||</sup>

$$C_D = C_f \left[ 3 \left( \frac{\ell}{d} \right) + 4.5 \left( \frac{d}{\ell} \right)^{1/2} + 21 \left( \frac{d}{\ell} \right)^2 \right].$$

We assume a friction coefficient of  $C_f = 0.0035$ . The drag is proportional to the surface area of the tank, which for an ellipsoid is

$$S = \frac{\pi}{2} d^2 \left( 1 + \frac{\ell}{de} \arcsin e \right)$$

Where  $e = \sqrt{1 + d^2/\ell^2}$ . The volume of the fuel tank is

$$V = \frac{\pi}{6} d^2 \ell,$$

and the required volume is  $V_{\text{req}} = 2.5 \text{ m}^3$ .

Find the optimum tank length and diameter numerically using your own optimizer or a software package. Verify your solution graphically by plotting the objective function contours and the constraint.

5.12 Solve a variation of Ex. 5.24 where we replace the system of cables with a cable and a rod that resists both tension and compression. The cable is positioned above the spring as shown in Fig. 5.36, where  $x_c = 2 \text{ m}$  and  $y_c = 3 \text{ m}$ , with a maximum length of  $\ell_c = 7.0 \text{ m}$ . The rod is positioned at  $x_r = 2 \text{ m}$  and  $y_r = 4 \text{ m}$ , with a length of  $\ell_r = 4.5 \text{ m}$ . How does this change the formulation of the problem? Does the optimum change?

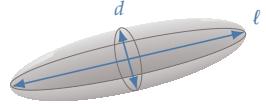


Figure 5.35: Ellipsoid fuel tank

<sup>||</sup> Hoerner<sup>74</sup> provides this approximation in page 6-18.

<sup>74</sup> Hoerner, *Fluid-Dynamic Drag*. 1965

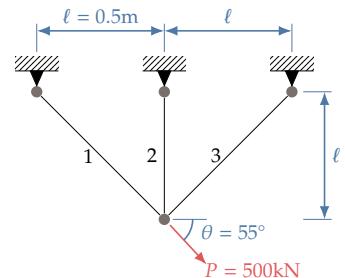
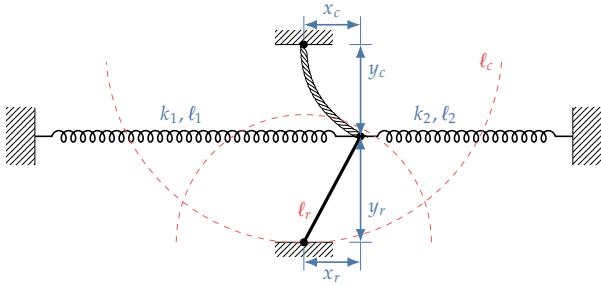


Figure 5.37: Three-bar truss elements



**Figure 5.36:** Spring system constrained by two cables.

5.13 *Three-Bar Truss.* Consider the truss shown in Fig. 5.37.<sup>\*\*</sup> The truss is subjected to a load  $P$  and we want to minimize the mass of the structure subject to stress and buckling constraints. The axial stresses in each bar are

$$\begin{aligned}\sigma_1 &= \frac{1}{\sqrt{2}} \left[ \frac{P \cos \theta}{A_o} + \frac{P \sin \theta}{A_o + \sqrt{2}A_m} \right] \\ \sigma_2 &= \frac{\sqrt{2}P \sin \theta}{A_o + \sqrt{2}A_m} \\ \sigma_3 &= \frac{1}{\sqrt{2}} \left[ \frac{P \sin \theta}{A_o + \sqrt{2}A_m} - \frac{P \cos \theta}{A_o} \right],\end{aligned}$$

where  $A_o$  is the cross-sectional area of the outer bars 1 and 3, and  $A_m$  is the cross-sectional area of the middle bar 2. The full optimization problem for the three-bar truss is

minimize	$\rho \left[ \ell(2\sqrt{2}A_1 + A_2) \right]$	
by varying	$A_o, A_m$	cross-sectional areas
subject to	$A_{\min} - A_o \leq 0$	minimum area
	$A_{\min} - A_m \leq 0$	minimum area
	$\sigma_{\text{yield}} - \sigma_1 \leq 0$	stress constraint for bar 1
	$\sigma_{\text{yield}} - \sigma_2 \leq 0$	stress constraint for bar 2
	$\sigma_{\text{yield}} - \sigma_3 \leq 0$	stress constraint for bar 3
	$-\sigma_1 - \frac{\pi^2 E \beta A_o}{2\ell^2} \leq 0$	buckling for bar 1
	$-\sigma_2 - \frac{\pi^2 E \beta A_m}{2\ell^2} \leq 0$	buckling for bar 2
	$-\sigma_3 - \frac{\pi^2 E \beta A_o}{2\ell^2} \leq 0$	buckling for bar 3

In the buckling constraints,  $\beta$  relates the second moment of area to the area ( $I = \beta A^2$ ) and is dependent on the cross-sectional shape

<sup>\*\*</sup>This is a well-known optimization problem formulated by Schmit<sup>25</sup> when he first proposed integrating numerical optimization with finite-element structural analysis

25. Schmit, *Structural Design by Systematic Synthesis*. 1960

of the bars. Assuming a square cross-section,  $\beta = 1/12$ . The bars are made out of an aluminum alloy with the following properties:  $\rho = 2710 \text{ kg/m}^3$ ,  $E = 69 \text{ GPa}$ ,  $\sigma_{\text{yield}} = 110 \text{ MPa}$ .

Find the optimal bar cross-sectional areas using your own optimizer or a software package. Which constraints are active? Verify your result graphically. *Exploration:* Try different combinations of unit magnitudes (e.g., Pa versus MPa for the stresses) for the functions of interest and the design variables to observe the effect of scaling.

- 5.14 Solve the same three-bar truss optimization problem of Prob. 5.13 by aggregating all the constraints into a single constraint. Try different aggregation parameters and see how close you can get to the solution you obtained for Prob. 5.13.
- 5.15 *Ten-bar Truss.* Consider the ten-bar truss structure described in Appendix C.2.2. The full design optimization problem is

$$\begin{aligned} & \text{minimize} && \rho \sum_{i=1}^{10} A_i \ell_i \\ & \text{by varying} && A_i, \quad i = 1, \dots, 10 && \text{cross-sectional areas} \\ & \text{subject to} && A_i \geq A_{\min} && \text{minimum area} \\ & && |\sigma_i| \leq \sigma_{y_i} \quad \text{for } i = 1, \dots, 10 && \text{yield stress constraints} \end{aligned}$$

Find the optimal mass and corresponding cross-sectional areas using your own optimizer or a software package.. Show a convergence plot. Report the number of function evaluations and the number of major iterations. *Exploration:* Restart from different starting points. Do you get more than one local minimum? What can you conclude about the multimodality of the design space?

- 5.16 Solve the same three-bar truss optimization problem of Prob. 5.15 by aggregating all the constraints into a single constraint. Try different aggregation parameters and see how close you can get to the solution you obtained for Prob. 5.15.

## Computing Derivatives

# 6

Derivatives play a central role in many numerical algorithms. In the context of optimization, we are interested in computing derivatives for the gradient-based optimization methods introduced in the previous chapter. The accuracy and computational cost of the derivatives is critical for the success of these optimization methods. In this chapter, we introduce the various methods for computing derivatives and discuss the relative advantages of each method.

By the end of this chapter you should be able to:

1. List the various methods used to compute derivatives.
2. Describe the pros and cons of these methods.
3. Use the methods in computational analyses.

### 6.1 Derivatives, Gradients, and Jacobians

The derivatives we focus on are *first-order* derivatives of one or more functions of interest ( $f$ ) with respect to a vector of variables ( $x$ ). In the engineering optimization literature, the term “sensitivity analysis” is often used to refer to the computation of derivatives, and derivatives are sometimes referred to as “sensitivity derivatives” or “design sensitivities.” While these terms are not incorrect, we prefer to use the more specific and concise term, *derivative*.

For the gradient-based, unconstrained methods introduced in the previous chapter, we need the gradient of the objective (a scalar) with respect to the vector of variables, which is a column vector with  $n_x$  components:

$$\nabla f(x) = \left[ \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_{n_x}} \right]^T. \quad (6.1)$$

In general, however, optimization problems in engineering design are constrained. We will see in Chapter 5 that for constrained, gradient-

based optimization, we also need the gradients of all the constraints with respect to all the design variables.

For the sake of generality, we do not distinguish between the objective and constraints in this chapter. Instead, we refer to the functions being differentiated as the *functions of interest*, and represent them as a vector-valued function,  $f = [f_1, f_2, \dots, f_{n_f}]^T$ . The derivatives of all the functions of interest with respect to all the variables form the *Jacobian matrix*,

$$J(x) = \begin{bmatrix} \nabla f_1^T \\ \vdots \\ \nabla f_{n_f}^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{n_f}}{\partial x_1} & \dots & \frac{\partial f_{n_f}}{\partial x_{n_x}} \end{bmatrix}, \quad (6.2)$$

which is an  $(n_f \times n_x)$  rectangular matrix where each row corresponds to the gradient of each function with respect to all the variables. Sometimes it is useful to write the Jacobian in index notation,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}. \quad (6.3)$$

Row  $i$  of the Jacobian is the gradient of function  $f_i$ . Each column in the Jacobian is called the *tangent* with respect to a given variable  $x_j$ .

---

**Example 6.1:** Jacobian of a vector-valued function.

Consider the following function with two inputs and two outputs:

$$f(x) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin x_1 \\ x_1 x_2 + x_2^2 \end{bmatrix}. \quad (6.4)$$

We can differentiate this symbolically to obtain exact reference values.

$$\frac{df}{dx} = \begin{bmatrix} x_2 + \cos x_1 & x_1 \\ x_2 & x_1 + 2x_2 \end{bmatrix}. \quad (6.5)$$

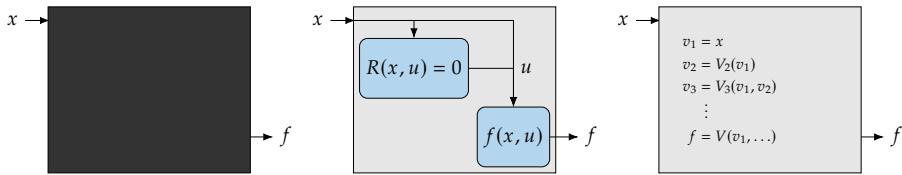
We evaluate this at  $x = (\pi/4, 2)$ , which yields:

$$\frac{df}{dx} = \begin{bmatrix} 2.707 & 0.785 \\ 2.000 & 4.785 \end{bmatrix}. \quad (6.6)$$


---

## 6.2 Overview of Methods for Computing Derivatives

The methods for computing derivatives can be classified according to the representation used for the numerical model. There are three possible representations, as shown in Fig. 6.1. In one extreme (left), we do not know anything about the model and consider it a black box where we only have control over the inputs and observe the outputs. When this is the case, we can only compute derivatives using finite differences (Section 6.4). In the other extreme (right), we have access to the all the source code used to compute the functions of interest and perform the differentiation line by line. This is the essence of the algorithmic differentiation approach (Section 6.6), as well as the complex-step method (Section 6.5). In the intermediate case we look at the model residuals and states (middle), which are the basis for the implicit analytic methods (Section 6.7). Finally, when the model can be represented with multiple components, we can use a coupled derivative approach where any of the above methods can be used for each component (Section 13.4).



**Tip 6.2:** Identify and fix the sources of numerical noise.

As mentioned in Tip 3.8, it is important to find out the level of numerical noise in your model. This is especially important when computing derivatives of the model because taking the derivative can amplify the noise. There are several common sources of model numerical noise, some of which can be mitigated. First, you should establish the level of numerical noise in your model (see Tip 3.8). Then you can do the same for the derivative and see if you get enough precision.

Iterative solvers can introduce numerical noise when the convergence tolerance is to high or when they have an inherit limit in their precision (see Section 3.5.3). When you do not have enough precision, you can try to reduce the convergence tolerance or increase the iteration limit.

Another possible source of error is file input and output. Many legacy codes are driven by reading and writing input and output files. However, the numbers in the files usually have far fewer digits than a double precision floating point number. The ideal solution is to modify the original code so that it can be called directly so that the data is passed in memory. Another solution

**Figure 6.1:** Derivative computation methods can consider three different levels of information: function values, model states, and lines of code.

is to change the output precision in the files.

---

**Tip 6.3:** Smooth model discontinuities.

Many models are defined in a piecewise manner, resulting in a discontinuous function value, discontinuous derivative, or both. This can happen even if the underlying physical behavior is continuous (for example, by using a non-smooth interpolation of experimental data). The solution is to modify the implementation so that it is continuous and still consistent with the physics. If the physics is truly discontinuous, it might still be advisable to artificially smooth the function as long as there is no serious degradation in the modeling error. Even if the smoothed version is highly nonlinear, having a continuous first derivative will help in the derivative computation and gradient-based optimization.

---

### 6.3 Symbolic Differentiation

Symbolic differentiation is well known and widely used in calculus, but it is of limited use in numerical optimization because it is only applicable for explicit functions. Except for the simplest cases (such as Ex. 6.1), many computational models are implicitly defined and involve iterative solvers (see Chapter 3). While the mathematical expression within these iterative procedures are explicit, it is challenging, or even impossible, to use symbolic differentiation to obtain closed-form mathematical expressions for the derivative of the procedure. Even when it is possible, these expressions are almost always computationally inefficient.

---

**Example 6.4:** Difficulties associated with symbolic differentiation.

Kepler's equation describes the orbit of a body under gravity as briefly discussed in Chapter 2. Following Prob. 6.6 we seek to compute the quantity  $f$ , which is the difference between the eccentric and mean anomalies. For simplicity, we set the eccentricity to 1, and call the mean anomaly  $x$ . We are then left with the equation:

$$f = \sin(x + f) \quad (6.7)$$

We can see that  $f$  is an implicit function of  $x$ . As a simple numerical procedure, to determine the value of  $f$  for a given input  $x$ , we use fixed point iteration. That means that we start with a guess for  $f$ , input the value of  $f$  on only the right hand side of that expression to estimate a new value for  $f$ , and repeat.

In this case, convergence typically happens in about 10 iterations. Arbitrarily we choose  $x$  as the initial guess for  $f$  and the computational procedure is as follows:

```
Input: x
f = x
for i = 1 to 10 do
    f = sin(x + f)
end for
return f
```

Now that we have a computational procedure, we would like to compute the derivative  $df/dx$ . A symbolic math toolbox can be used to find the closed-form expression for this derivative as shown in Fig. 6.2, but the expression is extremely long and is full of redundant calculations. This problem becomes exponentially worse as you increase the number of iterations in the loop. Therefore, this approach is intractable for computational models of even moderate complexity—this is known as *expression swell*.

```
dfdx =
cos(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin(x +
    sin(2*x))))))))*(cos(x + sin(x + sin(x + sin(x + sin(x +
    sin(x + sin(x + sin(2*x))))))))*(cos(x + sin(x + sin(x + sin(x +
    sin(x + sin(x + sin(x + sin(2*x))))))))*(cos(x + sin(x + sin(x + sin(x +
    sin(x + sin(x + sin(x + sin(2*x))))))))*(cos(x + sin(x + sin(x + sin(x +
    sin(x + sin(2*x)))))))*(cos(x + sin(x + sin(x + sin(x + sin(2*x))))))
*(cos(x + sin(x + sin(x + sin(2*x)))))*(cos(x + sin(x + sin(2*x)))*(
cos(x + sin(2*x))*(2*cos(2*x) + 1) + 1) + 1) + 1) + 1) + 1)
```

Figure 6.2: Symbolic derivative of the simple function shown above.

---

Nevertheless, symbolic differentiation is still useful to derive derivatives of simple explicit components within a larger model. Furthermore, algorithm differentiation (discussed in a later section) relies on symbolic differentiation to differentiate each line of code in the model.

## 6.4 Finite Differences

Finite-difference methods are widely used to compute derivatives due to their simplicity. They are versatile because they require nothing more than function values. Finite-differences are the only viable option when dealing with “black-box” functions because they do not require any knowledge about how the function is evaluated. Most gradient-based optimization algorithms perform finite-differences by default when the user does not provide the required gradients. However, finite differences are neither accurate nor efficient.

### 6.4.1 Finite-Difference Formulas

Finite-difference approximations are derived by combining Taylor series expansions. Using the right combinations of these expansions, it is possible to obtain finite-difference formulas that estimate an arbitrary order derivative with any order of truncation error. The simplest finite-difference formula can be derived directly from a Taylor series expansion in the  $j^{\text{th}}$  direction,

$$f(x + h\hat{e}_j) = f(x) + h \frac{\partial f}{\partial x_j} + \frac{h^2}{2!} \frac{\partial^2 f}{\partial x_j^2} + \frac{h^3}{3!} \frac{\partial^3 f}{\partial x_j^3} + \dots, \quad (6.8)$$

where  $\hat{e}_j$  is the unit vector in the  $j^{\text{th}}$  direction. Solving the above for the first derivative we obtain the finite-difference formula,

$$\frac{\partial f}{\partial x_j} = \frac{f(x + h\hat{e}_j) - f(x)}{h} + O(h), \quad (6.9)$$

where  $h$  is the *finite-difference step size*. This approximation is called the *forward difference* and is directly related to the definition of a derivative since,

$$\frac{\partial f}{\partial x_j} = \lim_{h \rightarrow 0} \frac{f(x + h\hat{e}_j) - f(x)}{h} \approx \frac{f(x + h\hat{e}_j) - f(x)}{h}. \quad (6.10)$$

The truncation error is  $O(h)$ , and therefore this is a first-order approximation. The difference between this approximation and the exact derivative is illustrated in Fig. 6.3.

The backward difference approximation can be obtained by replacing  $h$  with  $-h$  to yield,

$$\frac{\partial f}{\partial x_j} = \frac{f(x) - f(x - h\hat{e}_j)}{h} + O(h), \quad (6.11)$$

which is also a first order approximation.

Assuming each function evaluation yields the full vector  $f$ , the above formulas compute the  $j^{\text{th}}$  column of the Jacobian (6.2). To compute the full Jacobian, we need to loop through each direction  $\hat{e}_j$ , add a step, recompute  $f$ , and compute a finite difference. Hence, the cost of computing the complete Jacobian is proportional to the number of input variables of interest,  $n_x$ .

For a second-order estimate of the first derivative, we can use the expansion of  $f(x - h\hat{e}_j)$  to obtain,

$$f(x - h\hat{e}_j) = f(x) - h \frac{\partial f}{\partial x_j} + \frac{h^2}{2!} \frac{\partial^2 f}{\partial x_j^2} - \frac{h^3}{3!} \frac{\partial^3 f}{\partial x_j^3} + \dots \quad (6.12)$$

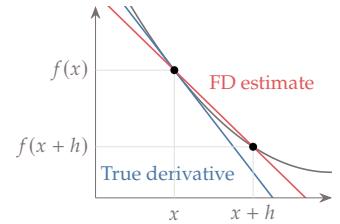


Figure 6.3: Exact derivative compared to a forward-difference finite-difference approximation.

Then, if we subtract this from the expansion (6.8) and solve the resulting equation for the derivative of  $f$ , we get the *central-difference* formula,

$$\frac{\partial f}{\partial x_j} = \frac{f(x + h\hat{e}_j) - f(x - h\hat{e}_j)}{2h} + O(h^2). \quad (6.13)$$

Even more accurate estimates can also be derived by combining different Taylor series expansions to obtain higher order truncation error terms. However, finite-precision arithmetic eventually limits the achievable accuracy (as will be discussed in the next section). With double precision arithmetic there are not enough significant digits to realize a significant advantage beyond central-difference.

We can also estimate second derivatives (or higher) by nesting finite-difference formulas. We can use, for example, the central difference formula (6.13) to estimate the second derivative instead of the first,

$$\frac{\partial^2 f}{\partial x_j^2} = \frac{\frac{\partial f}{\partial x_j}\Big|_{x+h\hat{e}_j} - \frac{\partial f}{\partial x_j}\Big|_{x-h\hat{e}_j}}{2h} + O(h^2). \quad (6.14)$$

Then we can use central difference again to estimate both  $f'(x + h)$  and  $f'(x - h)$  in the above equation to obtain,

$$\frac{\partial^2 f}{\partial x_j^2} = \frac{f(x + 2h\hat{e}_j) - 2f(x) + f(x - 2h\hat{e}_j)}{4h^2} + O(h). \quad (6.15)$$

The finite-difference method can also be used to compute directional derivatives, which represent the projection of the gradient into a given direction. To do this, instead of stepping in orthogonal directions to get the gradient, we need to step in the direction of interest,  $p$ . Using the forward difference, for example,

$$D_p f = \frac{f(x + hp) - f(x)}{h} + O(h). \quad (6.16)$$

One application of directional derivatives is to compute the slope in line searches (Section 4.3).

#### 6.4.2 The Step-Size Dilemma

When estimating derivatives using finite-difference formulas we are faced with the *step-size dilemma*. Because each estimate has a truncation error of  $O(h)$  (or  $O(h^2)$  when second order) we would like to choose as small of a step size as possible to reduce this error. However, as the step size reduces, *subtractive cancellation* (a finite-precision arithmetic error)

becomes dominant. In Table 6.1, for example, we show a case where we have 16-digit precision, and the step size was made small enough that the reference value and the perturbed value differ only in the last two digits. This means that when we do the subtraction required by the finite-difference formula, the final number only has two digits of precision. If  $h$  is made small enough, this difference vanishes to zero. If there were an infinite number of digits this wouldn't be a problem, but with finite precision arithmetic we are limited to larger step sizes.

$f(x + h)$	+1.234567890123431
$f(x)$	+1.234567890123456
$\Delta f$	-0.000000000000025

**Table 6.1:** Subtractive cancellation leads to a loss of precision and ultimately inaccurate finite difference estimates.

Theoretically, the optimal step size for a first-order finite difference is approximately  $\sqrt{\epsilon}$  where  $\epsilon$  is the precision of  $f$ . If we assume  $f$  is accurate to  $10^{-12}$  (meaning 12 digits of precision out of maximum of 15 for double precision), then the optimal step size is around  $10^{-6}$ . The error bound is also about  $\sqrt{\epsilon}$ , meaning that in this case, the finite difference would have about 6 digits of accuracy. Similarly, for central difference, the optimal step size scales approximately with  $\epsilon^{1/3}$ , with an error bound of  $\epsilon^{2/3}$ , meaning that for the previous example, the optimal step size would be around  $10^{-4}$  and it would achieve about 8 digits of accuracy. Even though 12 digits of accuracy would be expected based on the truncation error of  $O(h^2)$ , only a couple more digits of accuracy are realized because of finite-precision arithmetic. This step and error bound estimate above are just approximate and assume well-scaled problems.

Finite-difference approximations are sometimes used with larger steps than would be desirable from an accuracy standpoint in order to help smooth out numerical noise or discontinuities in the model. This type of approach can sometimes work, but it is better to address these problems within the model whenever possible.

---

**Tip 6.5:** When using finite-differencing, always perform a step-size study.

In practice, most gradient-based optimizers use finite-differences by default to compute the gradients. Given the potential for inaccuracies, finite differences are often the culprit in cases where gradient-based optimizers fail to converge. Although some of these optimizers try to estimate a good step size, there is no substitute for a step-size study by the user. The step-size study must be performed for all variables and does not necessarily apply to the whole design

space. Therefore, repeating this study for other values of  $x$  might be required.

---

### 6.4.3 Practical Implementation

A procedure for computing a Jacobian using forward finite-differences is detailed in Alg. 6.6. It is generally helpful to scale the step size based on the value of  $x_i$  (e.g.,  $h_i = 10^{-6}x_i$ ), unless  $10^{-6}x_i$  is already smaller than the default step size. Although the absolute step size usually differs for each  $h_i$ , the relative step size  $h$  is often the same, though that isn't necessary.

---

**Algorithm 6.6:** Forward finite-difference computation of the gradients of a vector-valued function  $f(x)$

**Inputs:**

$x$ : Point about which to compute the gradient  
 $f$ : Function of interest

**Outputs:**

$J$ : Jacobian of  $f$  about point  $x$

---

$f_0 = f(x)$	Evaluate reference values
$J = \text{matrix}(n_f, n_x)$	Initialize Jacobian as $n_f \times n_x$ matrix
$h = 10^{-6}$	Default relative step size. Optional input parameter(s)
<b>for</b> $j = 1$ to $n_x$ <b>do</b>	
$\Delta x = \max(h, h x_j)$	Scaled step size, but not smaller than $h$
$x_j += \Delta x$	Modification in place for efficiency although copying vector is an option
$f_+ = f(x)$	Evaluate function at perturbed point
$J_{*,j} = (f_+ - f_0) / \Delta x$	Fill one column of Jacobian at a time
$x_j -= \Delta x$	Do not forget to reset!
<b>end for</b>	

---

We can also use a directional derivative in arbitrary directions to verify the gradient computation. The directional derivative is the projection of the gradient in the chosen direction, i.e.,  $\nabla f^T p$ . This can be used to verify the gradient computed by some other method and is especially useful when the evaluation of  $f$  is expensive and computing the complete gradient is time consuming. We can verify a gradient by projecting it into some direction (say  $p = [1, \dots, 1]$ ), and then comparing it to the directional derivative in that direction. If the result matches the reference, then all the gradient components are most likely correct (you might try a couple more directions just to be sure).

However, if the result does not match, this directional derivative does not tell you which components of the gradient are incorrect.

## 6.5 Complex Step

The complex-step derivative approximation, strangely enough, computes derivatives of real functions using complex variables. Unlike finite differences, the complex-step method requires access to the source code, and therefore cannot be applied to black box components. The complex-step method is accurate, but no more efficient than finite-differences, as the computational cost still scales linearly with the number of variables.

This method originated with the work of Lyness<sup>75</sup> and Lyness *et al.*<sup>76</sup>, who developed formulas that use complex arithmetic for computing derivatives of real functions of arbitrary order with arbitrary order truncation error, much like the Taylor series combination approach in finite differences.

The simplest of the formulas approximate the first derivative using only one complex function evaluation and can be derived using a Taylor series expansion. Rather than using a real step  $h$ , as we did for the derivation of the finite-difference formulas, we use a pure imaginary step,  $ih$ . If  $f$  is a real function in real variables, and is also analytic, we can expand it in a Taylor series about a real point  $x$  as follows,

$$f(x + ih\hat{e}_j) = f(x) + ih \frac{\partial f}{\partial x_j} - \frac{h^2}{2} \frac{\partial^2 f}{\partial x_j^2} - i \frac{h^3}{6} \frac{\partial^3 f}{\partial x_j^3} + \dots \quad (6.17)$$

Taking the imaginary parts of both sides of this equation,

$$\text{Im}[f(x + ih\hat{e}_j)] = h \frac{\partial f}{\partial x_j} - \frac{h^3}{6} \frac{\partial^3 f}{\partial x_j^3} + \dots \quad (6.18)$$

Dividing this by  $h$  and solving for  $\partial f / \partial x_j$  yields the *complex-step derivative approximation*,

$$\frac{\partial f}{\partial x_j} = \frac{\text{Im}[f(x + ih\hat{e}_j)]}{h} + O(h^2), \quad (6.19)$$

which is a second order approximation. To use this approximation, you must provide a complex number with a perturbation in the imaginary part, compute the original function using complex arithmetic, and take the imaginary part of the output to obtain the derivative.

In practical terms, this means that we must convert the function evaluation so that it can take complex numbers as inputs and compute the corresponding complex outputs. Because we have assumed that

75. Lyness, *Numerical Algorithms Based on the Theory of Complex Variable*. 1967

76. Lyness *et al.*, *Numerical Differentiation of Analytic Functions*. 1967

$f(x)$  is a real function of a real variable, this most easily works with models that do not already involve complex numbers, but the procedure can be extended to work with functions that are already complex (multicomplex step),<sup>77</sup> or to provide exact second derivatives.<sup>78</sup> In Tip 6.10 we explain how to convert programs to handle the required complex arithmetic for the complex-step method to work in general.

Unlike finite-differences, this formula has no subtraction operation and thus no subtractive cancellation error. The only source of numerical error is the truncation error. However, if  $h$  is decreased to a small enough value (say,  $10^{-40}$ ), the truncation error can be eliminated. Then, the precision of the complex-step derivative approximation (6.19) will match the precision of  $f$ . This is a tremendous advantage over the finite-difference approximations (6.9) and (6.13).

Like the finite-difference approach, each evaluation yields a column of the Jacobian ( $\partial f / \partial x_j$ ), and the cost of computing all the derivatives is proportional to the number of design variables. The cost of the complex-step method is more comparable to that of a central difference as opposed to a forward difference because we must now compute a real and an imaginary part for every number in our program.

If we take the real part of the Taylor series expansion (6.17), we obtain the value of the function on the real axis,

$$f(x) = \operatorname{Re} [f(x + ih\hat{e}_j)] + O(h^2). \quad (6.20)$$

Similarly to the derivative approximation, we can make the truncation error disappear by using a small enough  $h$ . This means that no separate evaluation of  $f(x)$  is required to get the original real value of the function, we can simply take the real part of the complex evaluation.

What is a “small enough  $h$ ?” For the real function value (6.20) the truncation error is eliminated if  $h$  is such that

$$h^2 \left| \frac{1}{2} \frac{\partial^2 f}{\partial x_j^2} \right| < \epsilon |f(x)|, \quad (6.21)$$

where  $\epsilon$  is the relative working precision of the function evaluation. Similarly, for the derivative approximation we require

$$h^2 \left| \frac{1}{6} \frac{\partial^3 f}{\partial x_j^3} \right| < \epsilon |f'(x)|. \quad (6.22)$$

Although the step,  $h$ , can be set to an extremely small value, it is not always possible to satisfy this condition especially when the  $f$  or  $\partial f / \partial x_j$  tend to zero for the above two conditions, respectively. In practice, a step size in the  $10^{-20}$ – $10^{-40}$  range typically suffices.

77. Lantoine et al., *Using Multicomplex Variables for Automatic Computation of High-Order Derivatives*. 2012

78. Fike et al., *The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations*. 2011

The complex-step method can be used even when the evaluation of  $f$  involves the solution of numerical models through computer programs. The outer loop for computing the derivatives of multiple functions with respect to all variables (Alg. 6.7) is similar to the one for finite-differences. A reference function evaluation is not required, but now the function must be able to handle complex numbers correctly.

---

**Algorithm 6.7:** Computing the gradients of a vector-valued function  $f(x)$  using the complex-step method

**Inputs:**

$x$ : Point about which to compute the gradient

$f$ : Function of interest

**Outputs:**

$J$ : Jacobian of  $f$  about point  $x$

---

$J = \text{matrix}(n_f, n_x)$	<i>Initialize Jacobian as <math>n_f \times n_x</math> matrix</i>
$h = 10^{-40}$	<i>Typical “small enough” step size</i>
<b>for</b> $j = 1$ to $n_x$ <b>do</b>	
$x_j += \text{complex}(0, h)$	<i>Modify in place; could copy vector instead</i>
$f_+ = f(x)$	<i>Evaluate function perturbed with complex step</i>
$J_{*,j} = \text{imag}(f_+) / h$	<i>Extract derivatives from imaginary part</i>
$x_j -= \text{complex}(0, h)$	<i>Do not forget to reset!</i>
<b>end for</b>	

---



---

**Tip 6.8:** Check the convergence of the imaginary part.

When the solver that computes  $f$  is iterative, it is important to change the convergence criterion so that it checks for the convergence of the imaginary part in addition to the existing check on the real part. The imaginary part, which contains the derivative information, usually lags relative to the real part in terms of convergence. Therefore, if the solver only checks for the real part, it might yield a derivative with a precision lower than the function value.

---

**Example 6.9:** Complex step accuracy compared to finite differences.

To show the how the complex-step method works, consider the following analytic function:

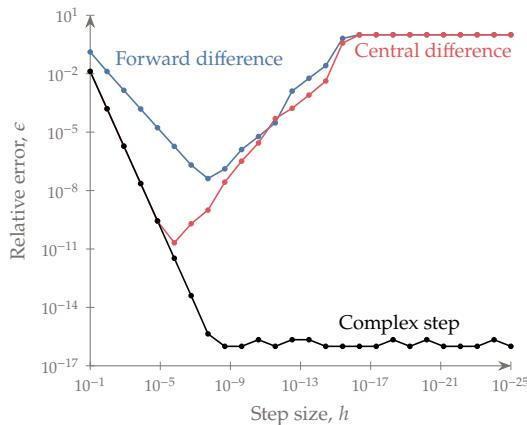
$$f(x) = \frac{e^x}{\sqrt{\sin^3 x + \cos^3 x}}. \quad (6.23)$$

The exact derivative at  $x = 1.5$  is computed to 16 digits based on symbolic differentiation as a reference value. The errors relative to this reference for

the complex-step derivative approximation and the forward and central finite-difference approximations are computed as

$$\epsilon = \frac{\left| \frac{df}{dx} \text{approx} - \frac{df}{dx} \text{exact} \right|}{\left| \frac{df}{dx} \text{exact} \right|} \quad (6.24)$$

As the step decreases, the forward-difference estimate initially converges at a linear rate, since its truncation error is  $O(h)$ , while the central-difference converges quadratically, as shown in Fig. 6.4. However, as the step is reduced below a value of about  $10^{-8}$  for the forward difference and  $10^{-5}$  for the central difference, subtractive cancellation errors become increasingly significant. When  $h$  is so small that no difference exists in the output (for steps smaller than  $10^{-16}$ ) the finite-difference estimates eventually yields zero (and  $\epsilon = 1$ ), which means 100% error.



**Figure 6.4:** Relative error of the derivative approximations as the step size decreases. Finite-difference approximations initially converge as the truncation error decrease, but when the step is too small, the subtractive cancellation errors become overwhelming. The complex-step approximation does not suffer from this issue. Note that the x-axis is oriented so that smaller step sizes are to the right.

The complex-step estimate converges quadratically with decreasing step size, as predicted by the truncation error term. The relative error reduces to machine zero around  $h = 10^{-8}$ , and stays there until  $h$  is so small that underflow occurs (around  $h = 10^{-308}$  in this case.)

Comparing the best accuracy of each of these approaches, we can see that by using finite-differences we only achieve a fraction of the accuracy that is obtained by using the complex-step approximation.

**Tip 6.10:** Code refactoring to handle complex step.

As previously mentioned the complex-step requires access to the source code of these programs. The reason is that changes are required to make sure that the program can handle complex numbers and the associated arithmetic,

that it handles logical operators consistently, and that certain functions yield the correct derivatives.

First, the program may need to be modified to use complex numbers. In programming languages like Fortran or C, this involves changing real valued type declarations (e.g., double) to complex type declarations (e.g., double complex). In some languages, such as Matlab, this is not necessary because functions are overloaded to accept either type automatically.

Second, some changes may be required to preserve the correct logical flow through the program. Relational logic operators such as “greater than” and “less than” or “if” and “else” are usually not defined for complex numbers. These operators are often used in programs, together with conditional statements, to redirect the execution thread. The original algorithm and its “complexified” version should follow the same execution thread, and therefore, defining these operators to compare only the real parts of the arguments is the correct approach. Functions that choose one argument, such as the maximum or the minimum values are based on relational operators. Following the previous argument, we should determine the maximum and minimum values based on the real parts alone.

Third, some functions need to be redefined for complex arguments. The most common function that needs to be redefined is the absolute value function, which for a complex number,  $z = x + iy$ , is defined as

$$|z| = \sqrt{x^2 + y^2}. \quad (6.25)$$

This definition is not complex analytic, which is required in the derivation of the complex-step derivative approximation. The following definition of absolute value,

$$\text{abs}(x + iy) = \begin{cases} -x - iy, & \text{if } x < 0 \\ +x + iy, & \text{if } x \geq 0, \end{cases} \quad (6.26)$$

yields the correct result since when  $y = h$ , the imaginary part divided by  $h$  corresponds to the slope of the absolute value function. There is an exception at  $x = 0$ , where the function is not analytic, but a derivative does not exist in any case. We use the “greater or equal” in the logic above so the approximation at least yields the correct right-sided derivative at that point.

Depending on the programming language, some trigonometric functions may also need to be redefined. This is because some default implementations, while correct, do not maintain accurate derivatives for small complex-step sizes using finite precision arithmetic. These must be replaced with mathematically equivalent implementations that avoid numerical issues.

Fortunately, most of these changes can be automated by using scripts to process the codes, and functions can be easily redefined using operator overloading in most programming languages. Martins *et al.*<sup>43</sup> provide more details on the problematic functions and how to implement the complex-step method in various programming languages.\*

43. Martins *et al.*, *The Complex-Step Derivative Approximation*. 2003

\*<https://goo.gl/TXqnpC>

## 6.6 Algorithmic Differentiation

Algorithmic differentiation (AD)—also known as computational differentiation or automatic differentiation—is a well known approach based on the systematic application of the differentiation chain rule to computer programs<sup>79,80</sup>. The derivatives computed with AD can match the precision of the function evaluation. We will see that the cost of computing derivatives with AD can be either proportional to the number of variables or to the number of functions, depending on the type of AD, which makes it flexible. Another attractive feature of AD is the fact that its implementation is largely automatic. To explain AD, we start by outlining the basic theory. We then show an example, and finally explain how the method is implemented in practice.

<sup>79.</sup> Griewank, *Evaluating Derivatives*. 2000

<sup>80.</sup> Naumann, *The Art of Differentiating Computer Programs—An Introduction to Algorithmic Differentiation*. 2011

### 6.6.1 Variables and Functions as Lines of Code

The basic concept of AD is straightforward. Even long, complicated codes consist of a sequence of basic operations (e.g., addition, multiplication, cosine, exponentiation). These operations are assembled in lines of code that compute variable values using explicit expressions, which can be differentiated symbolically with respect to all the variables in the expression. AD performs this symbolic differentiation and adds the code that computes the derivatives for each variable in the code. The derivatives of each variable are then computed using the chain rule.

The fundamental building blocks of a code are unary and binary operations. These operations can be combined to obtain more elaborate explicit functions, which are typically expressed in one line of computer code. We represent the variables in the computer code as the sequence  $v = v_1, \dots, v_i, \dots, v_N$ , where  $N$  is the total number of variables assigned in the code. One or more of these variables at the start of this sequence are given and correspond to  $x$ , while one or more of the variables toward the end of the sequence are the outputs of interest,  $f$ . In general, a variable assignment corresponding to a line of code can depend on any other variable, including itself, through a function  $V_i$  for line (or operation)  $i$ :

$$v_i = V_i(v_1, v_2, \dots, v_i, \dots, v_n), \quad (6.27)$$

where we adopt the convention that the lower case represents the *value* of the variable, and the upper case represents the *function* that computes that value. Except for the simplest codes, many of the variables in the code are overwritten due to iterative loops.

To understand AD, it is useful to imagine a version of the code where all the loops are *unrolled*. That is, instead of overwriting variables,

we just create new versions of those variables. Then, we can represent the computations in the code in a sequence with no loops such that each variable in this larger set only depends on *previous* variables, and then

$$v_i = V_i(v_1, v_2, \dots, v_{i-1}). \quad (6.28)$$

Given such a sequence of operations, and the derivatives for each operation, we can apply the chain rule to obtain the derivatives for the entire sequence. The chain rule can be applied in two ways. In the *forward mode*, we fix one input variable and work forward through the outputs until we get the desired total derivative. In the *reverse mode*, we fix one output variable and work backwards through the inputs until we get the desired total derivative.

### 6.6.2 Forward Mode AD

The chain rule for the forward mode can be written as:

$$\frac{dv_i}{dv_j} = \sum_{k=j}^{i-1} \frac{\partial V_i}{\partial v_k} \frac{dv_k}{dv_j}, \quad (6.29)$$

where  $V_i$  represents an explicit function. Using the forward mode, we evaluate a sequence of these expressions by fixing  $j$  (effectively choosing one input  $v_j$ ) and incrementing  $i$  to get the derivative of each variable  $v_i$ . We only need to sum up to  $i - 1$  because of the form (6.28), where each  $v_i$  only depends on variables that precede it. For a more convenient notation, we define a new variable that represents the derivative of variable  $i$  with respect to a fixed input  $j$  as:

$$\dot{v}_i \triangleq \frac{dv_i}{dv_j}. \quad (6.30)$$

Once we are done applying the chain rule (6.29) for the chosen input variable, we end up with the corresponding full column of the Jacobian, i.e., the tangent vector.

Suppose we have four variables:  $v_1, v_2, v_3, v_4$  and  $x = v_1, f = v_4$ , and we want  $df/dx$ . Using the above formula we set  $j = 1$  (as we want the derivative with respect to  $v_1 = x$ ) and increment in  $i$  to get the

sequence of derivatives

$$\begin{aligned}\dot{v}_1 &= 1 \\ \dot{v}_2 &= \frac{\partial V_2}{\partial v_1} \dot{v}_1 \\ \dot{v}_3 &= \frac{\partial V_3}{\partial v_1} \dot{v}_1 + \frac{\partial V_3}{\partial v_2} \dot{v}_2 \\ \dot{v}_4 &= \frac{\partial V_4}{\partial v_1} \dot{v}_1 + \frac{\partial V_4}{\partial v_2} \dot{v}_2 + \frac{\partial V_4}{\partial v_3} \dot{v}_3 \triangleq \frac{df}{dx}\end{aligned}\tag{6.31}$$

The colored derivatives show how the values are reused. In each step we just need to compute the partial derivatives of the current operation  $V_i$  and then multiply using total derivatives that have already been computed. We move forward evaluating partial derivatives of  $V$  in the same sequence that we evaluate the original function. This is convenient because all of the unknowns are *partial* derivatives, meaning that we only need to compute derivatives based on the operation at hand (or line of code).

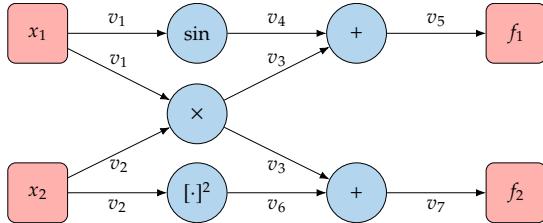
Using forward mode AD, obtaining derivatives with respect to additional outputs is either free (e.g.,  $dv_3/dv_1 \triangleq \dot{v}_3$  in Eq. 6.31) or requires only one more line of computation (e.g., if we had an additional output  $v_5$ ), and thus has a negligible additional cost for a large code. However, if we want the derivatives with respect to additional inputs (e.g.,  $dv_4/dv_2$ ) we would need to evaluate an entire set of similar calculations. Thus, the cost of the forward mode scales linearly with the number of inputs and is practically independent of the number of outputs.

---

**Example 6.11:** Forward mode AD for explicit functions.

Consider the function with two inputs and two outputs from Ex. 6.1. The explicit expressions in this function could be evaluated using only two lines of code. However, to make the AD process more apparent, we write the code such that each line has a single unary or binary operation, which is how a computer ends up evaluating the expression.

$$\begin{aligned}v_1 &= V_1(v_1) = x_1 \\ v_2 &= V_2(v_2) = x_2 \\ v_3 &= V_3(v_1, v_2) = v_1 v_2 \\ v_4 &= V_4(v_1) = \sin v_1 \\ v_5 &= V_5(v_3, v_4) = v_3 + v_4 = f_1 \\ v_6 &= V_6(v_2) = v_2^2 \\ v_7 &= V_7(v_3, v_6) = v_3 + v_6 = f_2\end{aligned}\tag{6.32}$$



**Figure 6.5:** Dependency graph for the numerical example evaluations (6.32).

The operations above result in the dependency graph shown in Fig. 6.5.

Say we want to compute  $d f_2 / d x_1$ , which in our example is  $d v_7 / d v_1$ . The evaluation point is the same as in Ex. 6.1:  $x = (\pi/4, 2)$ . Using the forward mode, set the seed for the corresponding input,  $\dot{v}_1$  to one, and the seed for the other input to zero. Then we get the sequence,

$$\begin{aligned}
 \dot{v}_1 &= 1 \\
 \dot{v}_2 &= 0 \\
 \dot{v}_3 &= \frac{\partial V_3}{\partial v_1} \dot{v}_1 + \frac{\partial V_3}{\partial v_2} \dot{v}_2 = v_2 \dot{v}_1 = 2 \\
 \dot{v}_4 &= \frac{\partial V_4}{\partial v_1} \dot{v}_1 = \cos v_1 \dot{v}_1 = 0.707\dots \\
 \dot{v}_5 &= \frac{\partial V_5}{\partial v_3} \dot{v}_3 + \frac{\partial V_5}{\partial v_4} \dot{v}_4 = \dot{v}_3 + \dot{v}_4 = 2.707\dots = \frac{\partial f_1}{\partial x_1} \\
 \dot{v}_6 &= \frac{\partial V_6}{\partial v_2} \dot{v}_2 = 2v_2 \dot{v}_2 = 0 \\
 \dot{v}_7 &= \frac{\partial V_7}{\partial v_3} \dot{v}_3 + \frac{\partial V_7}{\partial v_6} \dot{v}_6 = \dot{v}_3 + \dot{v}_6 = 2 = \frac{\partial f_2}{\partial x_1},
 \end{aligned} \tag{6.33}$$

where we have evaluated the partial derivatives numerically. We now have a *procedure* (not a symbolic expression) for computing  $d f_2 / d x_1$  for any  $(x_1, x_2)$ .

While we set out to compute  $d f_2 / d x_1$ , we also obtained  $d f_1 / d x_1$  as a byproduct. For a given input, we can obtain the derivatives for all outputs for essentially the same cost as one output. In contrast, if we want the derivative with respect to the other input,  $d f_1 / d x_2$ , a new sequence of calculations is necessary. Because this example contains so few operations the difference is small, but for a long program with many inputs, the difference will be large.

So far, we have assumed that we are computing Cartesian derivatives (i.e., derivatives with respect to each orthogonal component of  $x$ ). However, just like for finite differences, it is possible to compute directional derivatives using forward mode AD. This is done by setting the appropriate *seed* in the  $\dot{v}$ 's that correspond to the inputs in a vectorized manner. Suppose we have  $x \triangleq [v_1, \dots, v_{n_x}]^T$ . To compute the Cartesian derivative with respect to  $x_1$ , for example, we would set the seed as a vector  $\dot{v} = [1, 0, \dots, 0]^T$ , and similarly for the other

components. To compute a directional derivative in direction  $p$ , we would set the seed to the vector  $\dot{v} = p/\|p\|$ .

### 6.6.3 Reverse Mode AD

The *reverse mode* is also based on the chain rule, but using the alternative form:

$$\frac{dv_i}{dv_j} = \sum_{k=j+1}^i \frac{\partial V_k}{\partial v_j} \frac{dv_i}{dv_k}. \quad (6.34)$$

where the summation happens in reverse (starts at  $i$  and decrements to  $j + 1$ ). This is less intuitive than the forward chain rule, but it is mathematically valid. Here, we fix the index  $i$  corresponding to the output of interest and decrement  $j$  until we get the desired derivative. Similarly to the forward mode total derivative notation (6.30), we define a more convenient notation for the variables that carry the total derivatives with a fixed  $i$ ,

$$\bar{v}_j \triangleq \frac{dv_i}{dv_j}, \quad (6.35)$$

which are sometimes called *adjoint* variables. These reverse mode variables represent the derivatives of one output,  $i$ , with respect to all the variables, instead of the derivatives of all the variables with respect to one input,  $j$ , in the forward mode. Once we are done applying the reverse chain rule (6.34) for the chosen output variable, we end up with the corresponding full row of the Jacobian, i.e., the gradient vector.

Applying the reverse mode to the same four variable example as before, we get the following sequence of derivative computations (we set  $i = 4$  and decrement  $j$ ):

$$\begin{aligned} \bar{v}_4 &= 1 \\ \bar{v}_3 &= \frac{\partial V_4}{\partial v_3} \bar{v}_4 \\ \bar{v}_2 &= \frac{\partial V_3}{\partial v_2} \bar{v}_3 + \frac{\partial V_4}{\partial v_2} \bar{v}_4 \\ \bar{v}_1 &= \frac{\partial V_2}{\partial v_1} \bar{v}_2 + \frac{\partial V_3}{\partial v_1} \bar{v}_3 + \frac{\partial V_4}{\partial v_1} \bar{v}_4 \triangleq \frac{df}{dx} \end{aligned} \quad (6.36)$$

The partial derivatives for  $V$  must be computed for  $V_4$  first, then  $V_3$ , and so on. Therefore, we have to traverse the code in reverse. In practice, not every variable depends on every other variable, so a dependency graph is created during code evaluation. Then, when computing the adjoint derivatives we traverse the dependency graph in reverse. As before, the derivatives we need to compute in each line are only partial derivatives.

Using the reverse mode of AD, obtaining derivatives with respect to additional inputs is either free (e.g.,  $dv_4/dv_2 \triangleq \bar{v}_2$ ) or requires one more line of code to be evaluated. However, if we want the derivatives with respect to additional outputs (e.g.,  $dv_3/dv_1$ ) we would need to evaluate a different sequence of derivatives. Thus, the cost of the reverse mode scales linearly with the number of outputs and is practically independent of the number of inputs.

One complication with the reverse mode is that the resulting sequence of derivatives requires the values of the variables, starting with the last ones and progressing in reverse. Therefore, the code needs to run in a forward pass first and all the variables must be stored for use in the reverse pass, which has an impact on memory usage.

---

**Example 6.12:** Reverse mode AD.

To compute the same derivative using the reverse mode, we need to choose the output to  $f_2$  by setting the seed for the corresponding output,  $\bar{v}_7$  to one. Then we get,

$$\begin{aligned}
\bar{v}_7 &= 1 \\
\bar{v}_6 &= \frac{\partial V_7}{\partial v_6} \bar{v}_7 &= \bar{v}_7 &= 1 \\
\bar{v}_5 &= &= &= 0 \text{ (nothing depends on } v_5) \\
\bar{v}_4 &= \frac{\partial V_5}{\partial v_4} \bar{v}_5 &= \bar{v}_5 &= 0 \\
\bar{v}_3 &= \frac{\partial V_7}{\partial v_3} \bar{v}_7 + \frac{\partial V_5}{\partial v_3} \bar{v}_5 &= \bar{v}_7 + \bar{v}_5 &= 1 \\
\bar{v}_2 &= \frac{\partial V_6}{\partial v_2} \bar{v}_6 + \frac{\partial V_3}{\partial v_2} \bar{v}_3 &= 2v_2 \bar{v}_6 + v_1 \bar{v}_3 &= 4.785 = \frac{\partial f_2}{\partial x_2} \\
\bar{v}_1 &= \frac{\partial V_4}{\partial v_1} \bar{v}_4 + \frac{\partial V_3}{\partial v_1} \bar{v}_3 &= (\cos v_1) \bar{v}_4 + v_2 \bar{v}_3 &= 2 = \frac{\partial f_2}{\partial x_1},
\end{aligned} \tag{6.37}$$

While we set out to evaluate  $df_2/dx_1$ , we also computed  $df_2/dx_2$  as a byproduct. For each output, the derivatives of the all inputs come at the cost of evaluating only one more line of code. Conversely, if we want the derivatives of  $f_1$  a whole new set of computations is needed.

In the reverse computation sequence above, we needed the values of some of the variables in the original code to be precomputed and stored. In addition, the reverse computation requires the dependency graph information (which, for example, is how we would know that nothing depended on  $v_5$ ). In forward mode, the computation of a given derivative,  $\dot{v}_i$ , requires the partial derivatives of the line of code that computes  $v_i$  with respect to its inputs. In the reverse case, however, to compute a given derivative,  $\bar{v}_j$ , we require the partial derivatives of the functions that the current variable  $v_j$  affects with respect to  $v_j$ . Knowledge

of the function a variable affects is not encoded in that variable computation, and that is why the dependency graph if required.

---

Unlike forward mode AD and finite differences, it is not possible to compute a directional derivative by setting the appropriate seeds. While the seeds in the forward mode are associated with the inputs, the seeds for the reverse mode are associated with the outputs. Suppose we have multiple functions of interest,  $f \triangleq [v_{n-n_f}, \dots, v_n]^T$ . To find the derivatives of  $f_1$  in a vectorized operation, we would set  $\bar{v} = [1, 0, \dots, 0]^T$ . A seed with multiple nonzero components computes the derivatives of a *weighted* function with respect to all the variables, where the weight for each function is determined by the corresponding  $\bar{v}$  value.

In summary, if we want to compute a Jacobian matrix of partial derivatives of the outputs of interest with respect to all the inputs, the forward mode computes a column of derivatives in the Jacobian at each pass, while the reverse mode computes a row of the same Jacobian at each pass. Therefore, if we have more outputs (e.g., objective and constraints) than inputs (design variables) then the forward mode is more efficient. On the other hand, if we have many more inputs than outputs, then the reverse mode is more efficient. If the number of inputs is similar to the number of outputs, neither is particularly efficient and profiling the resulting code is necessary to determine which approach is faster. In practice, because of the need for creating a dependency graph the reverse mode is usually only favorable if the number of inputs is significantly larger than the number of outputs.

---

**Example 6.13:** Comparison of source code transformations and operator overloading.

There are two main ways to implement AD: by *source code transformation* or by using *derived datatypes and operator overloading*.

To implement AD by source transformation, the whole source code must be processed with a parser and all the derivative computations are introduced as additional lines of code. This approach is demonstrated below, using a forward mode, for the algorithm we used earlier to illustrate the difficulties of symbolic differentiation (Ex. 6.4). Differentiating this function with AD is much simpler. We set the seed,  $\dot{x}$ , to one, and for each function assignment we add the corresponding derivative line. As the loops are repeated,  $\dot{f}$  accumulates the derivative as  $f$  is successively updated.

**Input:**  $x, \dot{x}$

$$f = x$$

$$\dot{f} = \dot{x}$$

*Set seed  $\dot{x} = 1$  to get  $df/dx$*

```

for  $i = 1$  to  $20$  do
     $f = \sin(x + f)$ 
     $\dot{f} = (\dot{x} + \dot{f}) \cdot \cos(x + f)$ 
end for
return  $f, \dot{f}$ 

```

$df/dx$  is given by  $\dot{f}$

The reverse mode AD version of the same function is shown below. We set  $\bar{f} = 1$  to get the derivative of  $f$ . Now we need two distinct loops: a forward one that computes the original function, and a reverse one that accumulates the derivatives in reverse starting from the last derivative in the chain. Because the derivatives that are accumulated in the reverse loop depend on the intermediate values of the variables, we need to store all the variables in the forward loop. Here we do it via a stack, which is a data structure that stores a one-dimensional array. Using the stack concept, we can only add an element to the top of the stack (push) and take the element from the top of the stack (pop).

```

Input:  $x, \bar{f}$ 
for  $i = 1$  to  $20$  do
     $f = x$ 
    push( $f$ )
     $f = \sin(x + f)$ 
end for
for  $i = 20$  to  $1$  do
     $f = \text{pop}()$ 
     $\bar{f} = \cos(x + f) \cdot \bar{f}$ 
end for
 $\bar{x} = \bar{x} + \bar{f}$ 
return  $f, \bar{x}$ 

```

Set  $\bar{f} = 1$  to get  $df/dx$

*Put current value of  $f$  on top of stack*

*Do the reverse loop*

*Get value of  $f$  from top stack*

$df/dx$  is given by  $\bar{x}$

#### Tip 6.14: Operator overloading versus source code transformation.

To use derived types and operator overloading approach to AD, we need a language that supports these features, such as C++, Fortran 90, Python, Julia, Matlab, etc. The derived types feature is used to replace all the real numbers in the code,  $v$ , with a dual number type that includes both the original real number and the corresponding derivative as well, i.e.,  $u = (v, \dot{v})$ . Then, all operations are redefined (overloaded) such that, in addition to the result of the original operations, they yield the derivative of that operation. All these additional operations are performed “behind the scenes” without adding much code. Except for the variable declarations and setting the seed, the code remains exactly the same as the original.

There are AD tools available for most programming languages, including Fortran,<sup>81</sup> C/C++,<sup>82</sup> Python,<sup>83</sup> Julia,<sup>84</sup> and Matlab.<sup>85</sup> They have been extensively developed and provide the user with great functionality, including the calculation of higher-order derivatives, multivariable derivatives, and reverse mode options.<sup>†</sup>

<sup>81</sup> Hascoët *et al.*, TAPENADE 2.1 User’s Guide. 2004

<sup>82</sup> Griewank *et al.*, *Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*. 1996

<sup>83</sup> Wiltschko *et al.*, *Tangent: automatic differentiation using source code transformation in Python*. 2017

<sup>84</sup> Revels *et al.*, *Forward-Mode Automatic Differentiation in Julia*. 2016

<sup>85</sup> Neidinger, *Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming*. 2010

<sup>†</sup> Although some AD tools can be applied recursively to yield higher order derivatives, this approach is not typically efficient and is sometimes unstable.<sup>86</sup>

<sup>86</sup> Betancourt, *A geometric theory of higher-order automatic differentiation*. 2018

The operator overloading approach is much more elegant, since the original code stays practically the same and can be maintained directly. The source code transformation approach, on the other hand, enlarges the original code and results in code that is less readable, making it hard to debug the new extended code. Instead of maintaining source code transformed by AD, it is advisable to work with the original source, and devise a workflow where the parser is rerun before compiling a new version. The advantage of the source code transformation is that it tends to yield faster code, and it is easier to see what operations actually take place when debugging.

---

## 6.7 Implicit Analytic Methods—Direct and Adjoint

Direct and adjoint methods—which we refer to jointly as implicit analytic methods—linearize the model governing equations to obtain a system of linear equations where the derivatives are the unknown variables that can be solved for. Like the complex-step method and AD, implicit analytic methods can compute derivatives with a precision matching that of the function evaluation. Like AD, there are two main methods—direct and adjoint—that are analogous to the AD forward and reverse modes.

Analytic methods can be thought of as being in between the finite-difference method and AD in terms of the variables that they deal with. With finite differences, we only need to be aware of inputs and outputs, while AD involves dealing with every single variable assignment in the code. Analytic methods work at the model level, where we require knowledge of the governing equations and the corresponding state variables.

There are two main approaches to deriving implicit analytic methods: continuous and discrete. The continuous approach linearizes the original governing equations in PDE form, and then discretizes this continuous linearization. The discrete approach linearizes the discrete form of the governing equations instead. Each approach has its advantages and disadvantages. The discrete approach is preferred and is easier to generalize, so we explain the discrete approach exclusively.

### 6.7.1 Residuals and Functions

As mentioned in Chapter 3, a numerical model can be written as a set of residuals that need to be driven to zero,

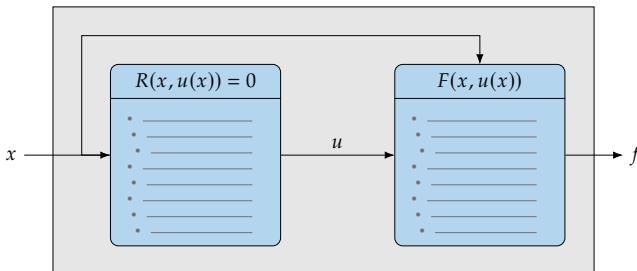
$$r = R(x, u(x)) = 0, \quad (6.38)$$

where these equations are solved for the  $n_u$  state variables  $u$  for a given fixed set of design variables  $x$ . This means that  $u$  is an *implicit* function of  $x$ .

The functions of interest,  $f$ , can in general be written as explicit functions of the state variables and the design variables, i.e.,

$$f = F(x, u(x)). \quad (6.39)$$

Therefore,  $f$  depends not only explicitly on the design variables, but also implicitly through the governing equations (6.38). This dependency is illustrated in Fig. 6.6.



**Figure 6.6:** Relationship between functions and design variables for a general set of implicit equations. The implicit equations  $r = 0$  define the states  $u$  for a given  $x$ , so the  $n_f$  functions of interest  $f$  end up depending both explicitly and implicitly on the  $n_x$  design variables  $x$ .

### 6.7.2 Direct and Adjoint Derivative Equations

The derivatives we ultimately want to compute are the ones in the Jacobian  $df/dx$ . Given the explicit and implicit dependence of  $f$  on  $x$ , we can use the chain rule to write the total derivative Jacobian of  $f$  as

$$\frac{df}{dx} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial u} \frac{du}{dx} \quad (6.40)$$

where the result is an  $(n_f \times n_x)$  matrix.

In this context, the total derivatives,  $df/dx$ , take into account the change in  $u$  that is required to keep the residuals of the governing equations, Eq. 6.38, equal to zero. The partial derivatives represent just the variation of  $f = F(x, u)$  with respect to changes in  $x$  or  $u$  without any regard to satisfying the governing equations. To better understand this, imagine computing these derivatives using finite differences. For the total derivatives, we would perturb  $x$ , solve the governing equations to obtain  $u$ , and then compute  $f$ , which would account for both dependency paths in Fig. 6.6. To compute the partial derivatives  $\partial F/\partial x$  and  $\partial F/\partial u$ , however, we would perturb  $x$  or  $u$  and just recompute  $f$  without solving the governing equations. By definition,  $du/dx$  is a total derivative because  $u$  depends on  $x$  through the governing equations, a total derivative that would be costly to compute using finite differences.

In general, partial derivative terms are easy to compute, while total derivatives are not.

The total derivative of the governing equations residuals (6.38) with respect to the design variables can also be derived using the chain rule. Furthermore, the total derivatives of the residuals must be zero if the governing equations are to remain satisfied, and we can write:

$$\frac{dr}{dx} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial u} \frac{du}{dx} = 0, \quad (6.41)$$

where  $dr/dx$  and  $du/dx$  are both  $(n_u \times n_x)$  matrices, and the Jacobian,  $\partial R/\partial u$ , is a square matrix of size  $(n_u \times n_u)$ .

We can visualize the requirement for the total derivative (6.41) to be zero in Fig. 6.7, which is a simplified representation of the set of points that satisfy the governing equations. In this case, it is just a line that maps a scalar  $x$  to a scalar  $u$ . In the general case, the governing equations map  $x \in \mathbb{R}^{n_x}$  to  $u \in \mathbb{R}^{n_u}$  and the set of points that satisfy the governing equations is a manifold in  $x$ - $u$  space. As a result, any change,  $dx$ , must be accompanied by the appropriate change,  $du$ , so that the governing equations are still satisfied. If we look at small perturbations about a feasible point and want to remain feasible, the variations of  $x$  and  $u$  are no longer independent, because the total derivative of the governing equation residuals (6.41) with respect to  $x$  must be zero.

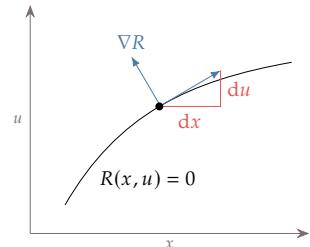
As in the total derivative of the function of interest (6.40), the partial derivatives here do not take into account the solution of the governing equations and are therefore much more easily computed than the total derivatives. However, if we provide the two partial derivative terms in the equation above, we can compute the total derivatives by solving the linear system, i.e.,

$$\frac{du}{dx} = - \left[ \frac{\partial R}{\partial u} \right]^{-1} \frac{\partial R}{\partial x}, \quad (6.42)$$

where the inverse of the  $\partial R/\partial u$  matrix does not necessarily mean that we actually find the inverse; rather, it represents the solution of the linear system using any suitable method (e.g., LU factorization). Since  $du/dx$  is a matrix with  $n_x$  columns, this linear system needs to be solved for each  $x$  with the corresponding column of the right-hand side matrix  $\partial R/\partial x$ . Substituting this result into the total derivative (6.40), we obtain:

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial u} \left[ \frac{\partial R}{\partial u} \right]^{-1} \frac{\partial R}{\partial x}, \quad (6.43)$$

where all the derivative terms on the right-hand side are partial derivatives. The partial derivatives in this equation can be computed using any



**Figure 6.7:** The residuals of the governing equations determine the values of  $u$  for a given  $x$ . Given a point that satisfies the equations, a perturbation in  $x$  about that point must be accompanied by the appropriate change in  $u$  for the equations to remain satisfied.

of the methods that we have described earlier: symbolic differentiation, finite difference, complex step, or AD.

The total derivative equation (6.43) shows that there are actually two ways to compute the total derivatives: the direct method and the adjoint method.

The *direct method* (which is already outlined above) consists in solving the linear system (6.42) and substituting  $du/dx$  into Eq. 6.40. We can express this by replacing the last two Jacobians with

$$\phi \triangleq \left[ \frac{\partial R}{\partial u} \right]^{-1} \frac{\partial R}{\partial x}. \quad (6.44)$$

Multiplying this by  $\partial R/\partial u$ , we get the linear system,

$$\frac{\partial R}{\partial u} \phi = \frac{\partial R}{\partial x}, \quad (6.45)$$

which we can solve to compute  $\phi$ . Then, we can replace  $\phi$  in the total derivative equation,

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial u} \phi. \quad (6.46)$$

This is also called the *forward mode* as it is analogous for forward mode AD. As previously mentioned, we need to solve the linear system (6.45) for one component of  $x$  at a time, and this equation has no dependence on any of the outputs  $F$ . Solving the linear system is the most computationally expensive operation in this procedure, and so the cost of this approach scales with the number of inputs  $n_x$ , but is essentially independent of the number of outputs  $n_f$ .

The *adjoint method* changes the linear system that needs to be solved to compute the total derivatives (6.43). Instead of solving the linear system with  $\partial R/\partial x$  as the right-hand side, we solve it with  $\partial F/\partial x$  in the right-hand side. This corresponds to replacing the two Jacobians in the middle by a new matrix of unknowns,

$$\psi^T \triangleq \frac{\partial F}{\partial u} \left[ \frac{\partial R}{\partial u} \right]^{-1}, \quad (6.47)$$

where the columns of  $\psi$ , are called the *adjoint vectors*. Multiplying by  $\partial R/\partial u$  on the right and taking the transpose of the whole equation, we get the *adjoint equations*,

$$\left[ \frac{\partial R}{\partial u} \right]^T \psi = \left[ \frac{\partial F}{\partial u} \right]^T. \quad (6.48)$$

This equation has no dependence on  $x$ . Again, this is the most expensive operation and so the cost of the adjoint method scales with the number

of outputs  $n_f$  and is essentially independent of the number of inputs  $n_x$ . Each adjoint vector is associated with a function of interest and is found by solving the linear system above with the corresponding row of  $\partial F/\partial u$ . The solution can then be used in the total derivative equations

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \psi^T \frac{\partial R}{\partial x}. \quad (6.49)$$

This is also called the reverse mode, as it is analogous to reverse mode AD.

The two approaches differ in cost, depending on the total derivatives that are required. The direct and adjoint methods require exactly the same partial derivative matrices, and the linear systems require the factorization of the same Jacobian matrix,  $\partial R/\partial u$ . The difference in cost between the two approaches boils down to how many times the corresponding linear system needs to be solved, as summarized in Table 6.2. The direct method requires a solution of the linear system (6.45) for each design variable in  $x$ , while the adjoint method requires a solution of the linear system (6.48) for each function of interest in  $f$ .

**Table 6.2:** Cost comparison of computing sensitivities for direct and adjoint methods.

Step	Direct	Adjoint
Partial derivative computation	Same	Same
Linear solution	$n_x$ times	$n_f$ times
Matrix multiplications	Same	Same

---

**Example 6.15:** Differentiating an implicit function.

Consider the following simplified equation for the natural frequency of a beam:

$$f = \lambda m^2 \quad (6.50)$$

where  $\lambda$  is a function of  $m$  through the following relationship

$$\frac{\lambda}{m} + \cos \lambda = 0 \quad (6.51)$$

Our goal is to compute the derivative  $df/dm$ , but  $\lambda$  is an implicit function of  $m$ . In other words, we cannot find an explicit expression for  $\lambda$  as a function of  $m$ , substitute that expression into Eq. 6.50, and then differentiate normally.

Fortunately, the direct and adjoint methods will allow us to compute this derivative.

Referring back to our nomenclature,

$$\begin{aligned} F(x, u(x)) &\triangleq F(m, \lambda(m)) = \lambda m^2, \\ R(x, u(x)) &\triangleq R(m, \lambda(m)) = \frac{\lambda}{m} + \cos \lambda = 0 \end{aligned} \quad (6.52)$$

where  $m$  is the design variable and  $\lambda$  is the state variable. The partial derivatives that we need to compute the total derivative (6.43) are:

$$\begin{aligned} \frac{\partial F}{\partial x} = \frac{\partial f}{\partial m} &= 2\lambda m, & \frac{\partial F}{\partial u} = \frac{\partial f}{\partial \lambda} &= m^2 \\ \frac{\partial R}{\partial x} = \frac{\partial R}{\partial m} &= -\frac{\lambda}{m^2}, & \frac{\partial R}{\partial u} = \frac{\partial R}{\partial \lambda} &= \frac{1}{m} - \sin \lambda \end{aligned} \quad (6.53)$$

Because this is a problem of only one function of interest and one design variable there is no distinction between the direct and adjoint methods (forward and reverse), and the matrix inverse is simply a division. Substituting these partial derivatives into the total derivative equation (6.43) yields:

$$\frac{df}{dm} = 2\lambda m + \frac{\lambda}{\frac{1}{m} - \sin \lambda}. \quad (6.54)$$

Thus, we are able to obtain the desired derivative in spite of the implicitly defined function. Here, it was possible to get an explicit expression for the total derivative, but in general, it is only possible to get a numeric value.

#### Example 6.16: Implicit analytic differentiation for two equations

Suppose we want to find the rectangle of a given area that is inscribed in an ellipse with given semi-major axes  $x_1, x_2$ , as shown in Fig. 6.8. The equation for the ellipse can be written as the residual,

$$r_1(u_1, u_2) = \frac{u_1^2}{x_1^2} + \frac{u_2^2}{x_2^2} - 1 = 0. \quad (6.55)$$

Of all the possible rectangles that can be inscribed in the ellipse, we want the rectangle with maximum area (spoiler alert: this is the solution for Prob. 5.5). Then, the area of the rectangle is constrained as,

$$r_2(u_1, u_2) = 4u_1u_2 - 2x_1x_2 = 0. \quad (6.56)$$

Suppose that our functions of interest are the rectangle perimeter and the rectangle aspect ratio,

$$f_1 = 4(u_1 + u_2), \quad f_2 = \frac{u_1}{u_2}. \quad (6.57)$$

We want to find the derivatives of these functions of interest with respect to the ellipse semi-major axes.

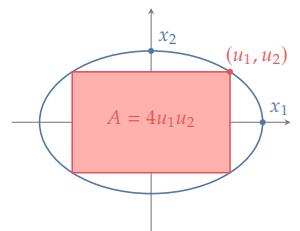


Figure 6.8: Rectangle inscribed in ellipse.

---

**Example 6.17:** Direct/adjoint methods applied to finite element analysis.

Now we consider a more complex example that applies to finite element structural analysis in general. The variables involved in finite element analysis map to the nomenclature as follows:

$x_i$  : design variables (element dimension, shape parameters)

$u_j$  : state variables (structural displacements)

$\mathcal{R}_k$  : residuals or governing equations

$f_n$  : functions of interest (mass, stress)

We use index notation for clarity because the equations involve derivatives of matrices with respect to vectors.

First, we need to derive expressions to compute all of the partial derivatives. The residual equations can be written as:

$$\mathcal{R}_k(x, u) = N_k(x, u) - F_k = 0 \quad (6.58)$$

where  $N$  is some nonlinear function and  $F$  is a vector of forces. Often, small displacements and elastic deformations are assumed, which leads to the linear form of the governing equations:

$$\mathcal{R}_k(x, u) = K_{kj}(x)u_j - F_k = 0 \quad (6.59)$$

where  $K$  is the stiffness matrix and we assumed that the external forces are not functions of the mesh node locations. The partial derivatives of these equations are:

$$\begin{aligned} \frac{\partial \mathcal{R}_k}{\partial x_i} &= \left[ \frac{\partial K_{kj}}{\partial x_i} \right] u_j \\ \frac{\partial \mathcal{R}_k}{\partial u_j} &= K_{kj}. \end{aligned} \quad (6.60)$$

The second equation is convenient because we already have the stiffness matrix. The first equation involves a new term, which are the derivatives of the stiffness matrix with respect to each of the design variables.

One of the functions of interest is the stress, which under the elastic assumption, is a linear function of the deflections that can be expressed as:

$$f_n(u) = \sigma_n(u) = S_{nj}u_j \quad (6.61)$$

Taking the partial derivatives,

$$\begin{aligned} \frac{\partial f_n}{\partial x_i} &= 0, \\ \frac{\partial f_n}{\partial u_j} &= S_{nj}. \end{aligned} \quad (6.62)$$

All of the partial derivatives above, except for one, require data that we already know. Putting everything together using the total derivative equation (6.43) yields,

$$\frac{d\sigma_n}{dx_i} = -S_{nj}K_{kj}^{-1} \left[ \frac{\partial K_{kj}}{\partial x_i} \right] u_j \quad (6.63)$$

We can solve this either using the direct or adjoint method. The direct method would solve the linear system for  $\phi$ , for one input  $i$  at a time, and then multiply through in the second equation:

$$\begin{aligned} K_{kj}\phi_j &= - \left[ \frac{\partial K_{kj}}{\partial x_i} \right] u_j \\ \frac{d\sigma_n}{dx_i} &= S_{nj}\phi_j \end{aligned} \quad (6.64)$$

This approach is preferable if we have few design variables  $x_i$  and many stresses  $\sigma_n$ . Or, we can solve this with the adjoint approach. The adjoint method solves the linear system for  $\psi$ , one output,  $n$ , at a time, and then multiplies through in the second equation:

$$\begin{aligned} K_{kj}\psi_k &= -S_{nj} \\ \frac{d\sigma_n}{dx_i} &= \psi_k \left[ \frac{\partial K_{kj}}{\partial x_i} \right] u_j. \end{aligned} \quad (6.65)$$

This approach is preferable when there are many design variables and few stresses.

---

## 6.8 Sparse Jacobians and Graph Coloring

In this chapter we have discussed various ways to compute a Jacobian. If the Jacobian is large and has many entries which are zero it is said to be *sparse*. In many cases we can take advantage of that sparsity to greatly accelerate the computational time required to construct the Jacobian.

When applying a forward mode, whether forward mode AD, finite differencing, or complex step, the cost of computing the Jacobian scales with  $n_x$ . Each forward pass completes one column of the Jacobian. For example, if using forward mode finite differencing,  $n_x$  evaluations would be required where, at iteration  $j$ , the input vector would be:

$$[x_1, x_2, \dots, x_j + h, \dots, x_{n_x}]^T \quad (6.66)$$

For many sparsity patterns, we can reduce computational cost greatly. As a motivating example, consider a square diagonal Jacobian:

$$\begin{bmatrix} J_{11} & 0 & 0 & 0 & 0 \\ 0 & J_{22} & 0 & 0 & 0 \\ 0 & 0 & J_{33} & 0 & 0 \\ 0 & 0 & 0 & J_{44} & 0 \\ 0 & 0 & 0 & 0 & J_{55} \end{bmatrix} \quad (6.67)$$

For this scenario, the Jacobian can be evaluated with one evaluation rather than  $n_x$  evaluations. This is because a given output  $f_i$  depends on only one input  $x_i$ . We could think of the outputs as  $n_x$  independent functions. Thus, for finite differencing rather than requiring  $n_x$  input vectors with  $n_x$  function evaluations we can use one input vector:

$$[x_1 + h, x_2 + h, \dots, x_j + h, \dots, x_{n_x} + h]^T \quad (6.68)$$

and compute all the nonzero entries in one shot.<sup>‡</sup>

While the diagonal case is perhaps easier to understand, it is fairly specialized. To continue the discussion more generally we will use the following  $5 \times 6$  matrix as an example:

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \quad (6.69)$$

<sup>‡</sup>Curtis *et al.*<sup>87</sup> were the first to show that the number of function evaluations could be reduced in evaluating the Jacobian for sparse matrices.

87. Curtis *et al.*, *On the Estimation of Sparse Jacobian Matrices*. 1974

A subset of columns that do not have more than one nonzero in the same row are said to have *structurally orthogonal* columns. For this example the following columns are structurally orthogonal: (1, 3), (1, 5), (2, 3), (2, 4, 5), (2, 6), and (4, 5). Structurally orthogonal columns can be combined, forming a smaller Jacobian that reduces the number of forward passes required. This reduced Jacobian is referred to as *compressed*. There is more than one way to compress the example Jacobian, but for this case the minimum number of compressed columns (referred to as *colors*) is three. A compressed Jacobian is shown below where columns 1 and 3 have been combined, and 2, 4, and 5 have been combined:

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \Rightarrow \begin{bmatrix} J_{11} & J_{14} & J_{16} \\ J_{23} & J_{24} & 0 \\ J_{31} & J_{32} & 0 \\ 0 & J_{45} & 0 \\ J_{53} & J_{55} & J_{56} \end{bmatrix} \quad (6.70)$$

For finite differencing or complex step, only compression amongst columns is possible. But for AD the reverse mode provides the opportunity to take advantage of compression along rows. The concept is the same, but instead we look for structurally orthogonal rows. One such compression is shown below.

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \Rightarrow \begin{bmatrix} J_{11} & 0 & 0 & J_{14} & J_{45} & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \quad (6.71)$$

AD can also be used even more flexibly where both modes are used: forward passes to evaluate groups of structurally orthogonal columns, and reverse passes to evaluate groups of structurally orthogonal rows. Rather than taking incremental steps in each direction as is done in finite differencing, in AD we set the seed vector with ones in the directions we wish to evaluate, similar to how the seed is set for directional derivatives as discussed in Section 6.6.

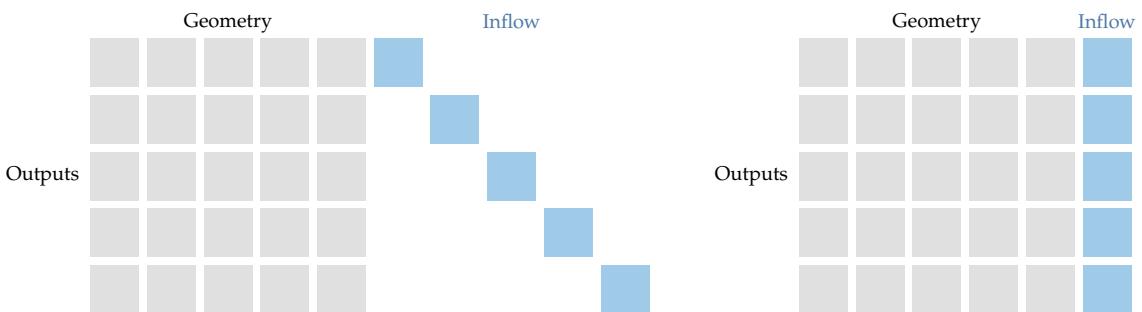
For these small Jacobians it is fairly straightforward to determine how best to compress the matrix. For a large matrix this is not so easy. The approach that is used is called *graph coloring*. In one approach, a graph is created with row and column indices as vertices and edges denoting nonzero entries in the Jacobian. Graph coloring algorithms use heuristics to estimate the fewest number of “colors” or orthogonal columns. Graph coloring is a large field of its own with derivative computation as just one application.<sup>§</sup>

---

**Example 6.18:** Speed up from sparse derivatives.
 

---

In static aerodynamic analyses the forces and moments produced at two different wind speeds are independent from each other, and if many different wind speeds are of interest, the resulting Jacobian will display a high degree of sparsity. Some examples include evaluating the power produced by a wind turbine across different wind speeds, or assessing an aircraft’s thrust throughout a flight envelope. Many other engineering analyses have similar structure. In this example we consider a typical blade optimization. The Jacobian is fully dense with respect to geometry changes, but as discussed is diagonal with respect to the various inflow conditions (left side of Fig. 6.9). We can compress the Jacobian as shown on the right side of Fig. 6.9.



To illustrate the potential benefits of using a sparse representation, the Jacobian was constructed for various sizes of inflow conditions using both forward AD, and forward AD with graph coloring (Fig. 6.10). After about 100 inflow conditions, the difference in time required exceeds an order of magnitude (note the log-log scale). As Jacobians are needed at every iteration

<sup>§</sup>Gebremedhin *et al.*<sup>88</sup> provide a review paper of graph coloring in the context of computing derivatives. Gray *et al.*<sup>89</sup> show how to use graph coloring to compute total coupled derivatives.

88. Gebremedhin *et al.*, *What Color Is Your Jacobian? Graph Coloring for Computing Derivatives*. 2005

89. Gray *et al.*, *OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization*. 2019

**Figure 6.9:** Representation of the Jacobian for this example. The blocks indicate areas where a derivative exists, and the blank spots where the derivative is always zero. The left is the original Jacobian and the right is the compressed representation.

in the optimization, this is a tremendous speedup, enabled just by leveraging the existing sparsity pattern.<sup>¶</sup>

<sup>¶</sup>The full details of this example are available in a preprint.<sup>90</sup>

90. Ning, *Using Blade Element Momentum Methods with Gradient-Based Design Optimization*, 2020

## 6.9 Unification of the Methods for Computing Derivatives

Now that we have introduced all the methods for computing derivatives, we will see how they are connected. For example, we have mentioned that the direct and adjoint methods are analogous to the forward and reverse mode of AD, respectively, but we have not formalized this relationship.

To get a broader view of these methods, we go back to the notion of the list of variables (6.27) considered when introducing AD:

$$v_i = V_i(v_1, v_2, \dots, v_i, \dots, v_n). \quad (6.72)$$

However, instead of defining these as every variable in a program, we are going to use a more flexible interpretation. At the very minimum, these variables include the inputs of interest,  $x$ , and the outputs of interest,  $f$ . The variables might also include intermediate variables, which we do not define for now. We are also going to use the concept of residuals that we already introduced, where,

$$r = R(v) = 0. \quad (6.73)$$

We use a more flexible interpretation of what these residuals are and they must be consistent with the definition of the variables: The number of residuals must be the same as the number of variables, and driving the residuals to zero must result in the correct variable values.

The *unified derivatives equation* (UDE) is based on these variables and residuals and be written as<sup>91</sup>:

$$\frac{\partial R}{\partial v} \frac{dv}{dr} = I = \frac{\partial R}{\partial v}^T \frac{dv}{dr}^T, \quad (6.74)$$

where  $I$  is the identity matrix and all the matrices are square and have the same size. The derivatives that we ultimately want to compute,  $df/dx$  are a subset of  $dv/dr$ . The matrix of partial derivatives needs to be constructed, and then the linear system is solved for the total derivatives. With the appropriate definition of the variables and the corresponding residuals (shown in Table 6.3), we can recover all the derivative computation methods using the UDE (6.74). The left-hand side represents forward (or direct) derivative computation, while the right-hand side represents reverse (or adjoint) derivative computation.

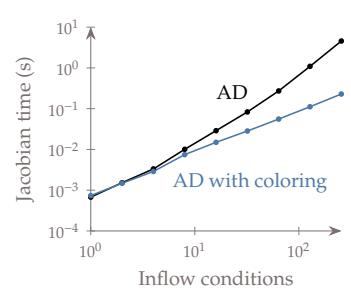


Figure 6.10: The compressed Jacobian.

91. Martins et al., *Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models*, 2013

For the inputs and outputs, the residuals assume that the associated variables ( $x$  and  $f$ ) are free, but they are constructed such that the variables assume the correct values when the residual equations are satisfied.

**Table 6.3:** Variable and residual definition needed to recover the various derivative computation methods with the UDE (6.74). The residuals of the governing equations are represented by  $R_g$  to distinguish them from the UDE residuals.

Method	Level	Variables, $v$	Residuals, $R$
Monolithic	Inputs and outputs	$\begin{bmatrix} x \\ f \end{bmatrix}$	$\begin{bmatrix} x - x_0 \\ f - F(x) \end{bmatrix}$
Analytic	Governing equations and state variables	$\begin{bmatrix} x \\ u \\ f \end{bmatrix}$	$\begin{bmatrix} x - x_0 \\ r - R_g(x, u) \\ f - F(x, u) \end{bmatrix}$
AD	Lines of code	$v$	$v - V(x, v)$

Using the variable and residual definitions from Table 6.3 for the monolithic method in the left hand side of the UDE (6.74), we get

$$\begin{bmatrix} I & 0 \\ -\frac{\partial F}{\partial x} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ \frac{df}{dx} & I \end{bmatrix} = I, \quad (6.75)$$

which yields the obvious result  $df/dx = \partial F/\partial x$ . This is not a particularly useful result, but it shows that the UDE can recover the monolithic case.

For the analytic derivatives, the left-hand side of the UDE becomes,

$$\begin{bmatrix} I & 0 & 0 \\ -\frac{\partial R_g}{\partial x} & -\frac{\partial R_g}{\partial u} & 0 \\ -\frac{\partial F}{\partial x} & -\frac{\partial F}{\partial u} & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ \frac{du}{dx} & \frac{du}{dr} & 0 \\ \frac{df}{dx} & \frac{df}{dr} & I \end{bmatrix} = I. \quad (6.76)$$

Since we are only interested in the  $df/dx$  block in the second matrix, we can ignore the second and third block columns of that matrix. Multiplying the remaining blocks out and using the definition  $\phi \triangleq -du/dx$ , we get the direct linear system (6.45) and the total derivative equation (6.46).

The right-hand side of the UDE yields the transposed system,

$$\begin{bmatrix} I & -\frac{\partial R_g^T}{\partial x} & -\frac{\partial F^T}{\partial x} \\ 0 & -\frac{\partial R_g^T}{\partial u} & -\frac{\partial F^T}{\partial u} \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & \frac{du}{dx} & \frac{df}{dx} \\ 0 & \frac{du}{dr} & \frac{df}{dr} \\ 0 & 0 & I \end{bmatrix} = I. \quad (6.77)$$

Similarly to the forward case, we ignore the block columns of the matrix of unknowns to focus on the block column containing  $df/dx$ . Multiplying this out, and defining  $\psi \triangleq -df/dr$ , we get the adjoint linear system (6.48) and the corresponding total derivative equation (6.49). The definition of  $\psi$  here is significant, since the adjoint vector can indeed be interpreted as the derivatives of the objective function with respect to the residuals of the governing equations.

Finally, we can recover AD from the UDE as well by defining the vector of variables as all the variables assigned in a code (with unrolled loops), and constructing the corresponding residuals. The forward mode yields

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ -\frac{\partial V_2}{\partial v_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ -\frac{\partial V_n}{\partial v_1} & \dots & -\frac{\partial V_n}{\partial v_{n-1}} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ \frac{dv_2}{dv_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ \frac{dv_n}{dv_1} & \dots & \frac{dv_n}{dv_{n-1}} & 1 \end{bmatrix} = I, \quad (6.78)$$

where the Jacobian  $df/dx$  is composed of a subset of derivatives in the corners near the  $dv_n/dv_1$  term. To compute these derivatives, we need to perform forward substitution and compute one column of the total derivative matrix at the time, where each column is associated with the inputs of interest.

The reverse mode yields

$$\begin{bmatrix} 1 & -\frac{\partial V_2}{\partial v_1} & \dots & -\frac{\partial V_n}{\partial v_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\frac{\partial V_n}{\partial v_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{dv_2}{dv_1} & \dots & \frac{dv_n}{dv_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & \frac{dv_n}{dv_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix} = I, \quad (6.79)$$

where the derivatives of interest are now near the top right corner of the total derivative matrix. To compute these derivatives, we need to perform back substitutions, which computes one column of the matrix at the time. Since the total derivative matrix is transposed here, the reverse mode actually computes a row of the total derivative Jacobian at the time, where each row is associated with an output of interest.

This is consistent with what we concluded before: The cost of the forward mode is proportional to the number of the inputs of interest, while the cost of the reverse move is proportional to the number of outputs of interest.

In this unification, we have found nothing new except for a new perspective on how all methods relate. This was achieved by generalizing the concept of “variable” and “residual.” As we will see later, these concepts and the UDE (6.74) have been used to develop a general framework for solving models and computing their derivatives<sup>37,89</sup>.

## 6.10 Summary

We discussed the methods available for computing derivatives. Each of these is summarized in the following list.

Symbolic differentiation is accurate, but only scalable for simple, explicit functions of low dimensionality. Although it cannot be used to derive a closed-form expression for models that are solved iteratively, symbolic differentiation is used by AD at each line of code, and can also be used in direct and adjoint methods to derive expressions for computing the required partial derivatives.

Finite difference formulas are popular because of their ease of use and because it is a black box method able to work with most any algorithm. The downsides are that they are not accurate and the cost scales linearly with the number of variables. Many of the issues optimization practitioners experience with gradient-based optimization can be traced to errors in the gradients when optimization algorithms automatically compute these gradients using finite differences.

The complex-step method is accurate and relatively easy to implement. It usually requires some changes to the analysis source code, but this process can be scripted. Its main advantage is that it produces analytically accurate derivatives. However, like the finite difference method, the cost scales linearly with the number of inputs, and each individual simulation requires almost twice the cost because of the use of complex arithmetic.

Algorithmic differentiation produces analytically accurate derivatives and can be scalable. Many implementations can be fully automated. The implementation requires access to the source code, but is still relatively straightforward to apply. Both forward and reverse modes are available, the former scales with the number of inputs and the latter scales with the number of outputs. The scaling factor for forward mode is generally much lower than finite differences, and in reverse mode is independent of the number of design variables.

Direct and adjoint methods are accurate, scalable, but require significant changes to source code. These methods are exact (depending on how the partial derivatives are obtained), and like algorithmic differentiation provides both forward and reverse modes with the same

37. Hwang et al., *A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives*. 2018

89. Gray et al., *OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization*. 2019

scaling advantages. The disadvantage is that the method is strongly intrusive and often considerable development effort is required.

## Problems

### 6.1 Answer *true* or *false* and justify your answer.

- a) A first-order derivative is only one of many types of sensitivity analysis.
- b) Each column of the Jacobian matrix represents the gradient of one of the functions of interest with respect to all the variables.
- c) You can only compute derivatives of models for which you have the source code, or at the very least understand how the model computes the functions of interest.
- d) Symbolic differentiation is intractable for all but the simplest models because of expression swell.
- e) Finite-difference approximations can compute first derivatives with a precision matching that of the function being differentiated.
- f) The complex-step method can only be used to compute derivatives of complex functions.
- g) Algorithmic differentiation uses a code parser to differentiate each line of code symbolically.
- h) The forward mode of algorithmic differentiation computes the derivatives of all outputs with respect to one input, while the reverse mode computes the derivative of one output with respect to all inputs.
- i) The adjoint method requires the same partial derivatives as the direct method.
- j) Of the two implicit analytic methods, the direct method is more widely used than the adjoint method because more problems have more design variables than functions of interest.
- k) Graph coloring makes Jacobians sparse by selectively replacing small-valued entries with zeros to trade accuracy for speed.
- l) The unified derivatives equation can represent implicit analytic and algorithmic differentiation approaches, but not monolithic differentiation.

- 6.2 Reproduce the comparison between the complex-step and finite-difference methods from Ex. 6.9. Reversing the  $x$ -axis as we did in Fig. 6.4 is not necessary. Do you get any complex-step derivatives with zero error compared to the analytic reference? What does that mean, and how should you show those points on the plot? Estimate the value of  $h$  required to eliminate truncation error in derivative using Eq. 6.22. Is this estimate consistent with your plot?
- 6.3 Compute the derivative using symbolic differentiation and using algorithmic differentiation (either forward or reverse mode) for the iterative code in Ex. 6.4. Use a package to facilitate the AD portion. Most scientific computing languages have AD packages.
- 6.4 Implement a forward-mode-AD tool using operator overloading to differentiate the function of Ex. 6.9. You need to define your own type and provide it with overloaded functions for  $\exp$ ,  $\sin$ ,  $\cos$ ,  $\sqrt$ , addition, division, and exponentiation.
- 6.5 Suppose you have two airplanes that are flying in a horizontal plane defined by  $x$  and  $y$  coordinates. Both airplanes start at  $y = 0$ , but airplane 1 starts at  $x = 0$  while airplane 2 has a head start of  $x = \Delta x$ . The airplanes fly at a constant velocity. Airplane 1 has a velocity  $v_1$  in the direction of the positive  $x$ -axis and airplane two has a velocity  $v_2$  at an angle  $\gamma$  with the  $x$ -axis. The functions of interest are the distance ( $d$ ) and the angle ( $\theta$ ) between the two airplanes as a function of time. The independent variables are  $\Delta x$ ,  $\gamma$ ,  $v_1$ ,  $v_2$ ,  $t$ . Write the code that computes the functions of interest (outputs) for a given set of independent variables (inputs). Use AD to differentiate the code. Choose a set of inputs, compute the derivatives of all the outputs with respect to the inputs and verify them against the complex-step method.
- 6.6 Kepler's equation, which we mentioned in Section 2.2, defines the relationship between a planet's polar coordinates and the time elapsed from a given initial point through the implicit equation,

$$E - e \sin(E) = M,$$

where  $M$  is the mean anomaly (a parameterization of time),  $E$  is the eccentric anomaly (a parameterization of the polar angle), and  $e$  is the eccentricity of the elliptical orbit. Suppose that the function of interest is the difference between the eccentric and mean anomalies,

$$f(E, M) = E - M.$$

Derive an analytic expression for  $df/de$  and  $df/dM$ . Verify your result against the complex-step method or AD (you will need a solver for Kepler's equation, which was the subject of Prob. 3.6).

6.7 Compute the derivatives for the ten-bar truss problem described in Appendix C.2.2 using the direct and adjoint implicit differentiation methods. We want to compute the derivatives of the objective (mass) with respect to the design variables (ten cross-sectional areas), and the derivatives of the constraints (stresses in all ten bars) with respect to the design variables (a  $10 \times 10$  Jacobian matrix). Compute the derivatives using:

- A finite-difference formula of your choice.
- The complex-step derivative method.
- Algorithmic differentiation.
- The implicit analytic method (direct and adjoint).

Report the errors relative to the implicit analytic methods. Discuss your findings and the relative merits of each approach.

6.8 We can now solve the ten-bar truss problem (previously solved in Prob. 5.15) using the derivatives computed in Prob. 6.7. Solve this optimization problem using both finite-difference derivatives and an implicit analytic method. Report the following:

- Convergence plot with two curves for the different derivative computation approaches on the same plot.
- Number of function calls required to converge for each method. This metric is more meaningful if you use more than one starting point and average the results.

Discuss your findings.

6.9 Aggregate the constraints for the ten-bar truss problem and extend the code from Prob. 6.7 to compute the required constraint derivatives using the implicit analytic method that is most advantageous in this case. Verify your derivatives against the complex-step method. Solve the optimization problem and compare your results to the ones you obtained in Prob. 6.8. How close can you get to the reference solution?

Gradient-free algorithms fill an important role in optimization. The gradient-based algorithms introduced in Chapter 4 are efficient in finding local minima for high-dimensional nonlinear problems defined by continuous smooth functions. However, the assumptions made for these algorithms are not always valid, which can render these algorithms ineffective. Also, gradients might not be available, as in the case of functions given as a black box.

In this chapter, we introduce only a few popular representative gradient-free algorithms. Most are designed to handle unconstrained functions only, but they can be adapted to solve constrained problems by using the penalty or filtering methods introduced in Chapter 5. We start by discussing the problem characteristics that are relevant to the choice between gradient-free and gradient-based algorithms and then give an overview of the types of gradient-free algorithms.

By the end of this chapter you should be able to:

1. Identify problems that are well-suited for gradient-free optimization.
2. Describe the characteristics and approach of more than one gradient-free optimization method.
3. Use gradient-free optimization algorithms to solve real engineering problems.

### 7.1 Relevant Problem Characteristics

Gradient-free can be useful when gradients are not available, such as when dealing with black-box functions. Although gradients can always be approximated with finite differences, these approximations suffer from potentially large inaccuracies (see Section 6.4.2). Gradient-based algorithms require a more experienced user because they take more effort to setup and run. Overall, gradient-free algorithms are easier

to get up and running but are much less efficient, particularly as the dimension of the problem increases.

One major advantage of gradient-free algorithms is that they do not assume function continuity. For gradient-based algorithms, function smoothness is essential when deriving the optimality conditions, both for unconstrained functions and constrained functions. More specifically, the KKT conditions (5.13) require that the function be continuous in value ( $C^0$ ), gradient ( $C^1$ ), and Hessian ( $C^2$ ) in at least a small neighborhood of the optimum. If, for example, the gradient is discontinuous at the optimum, it is undefined and the KKT conditions are not valid. Away from optimum points, this requirement is not as stringent. While gradient-based algorithms work on the same continuity assumptions, they can usually tolerate the occasional discontinuity as long as it is away from an optimum point. However, for functions with excessive numerical noise and discontinuities, gradient-free algorithms might be the only option.

Many considerations are involved when choosing between a gradient-based and a gradient-free algorithm. Some of these considerations are common sources of misconception. One problem characteristic that is often cited as a reason for choosing gradient-free methods is multimodality. Design space multimodality can be due to an objective function with multiple local minima, or in the case of a constrained problem, the multimodality can arise from the constraints that define disconnected or nonconvex feasible regions.

As we will see shortly, some gradient-free methods feature a global search that increases the likelihood of finding the global minimum. This feature is a reason why gradient-free methods are often used for multimodal problems. However, not all gradient-free methods are global search methods, some perform only a local search. Additionally, even though gradient-based methods are by themselves local search methods, they are often combined with global search strategies as discussed in Tip 4.24. It is not necessarily true that a global search, gradient-free method is more likely to find a global optimum than a multistart gradient-based method. As always, problem-specific testing is needed.

Furthermore, it is assumed far too often that any complex problem is multimodal, but that is often not the case. While it might be impossible to prove that a function is unimodal, it is easy to prove that a function is multimodal by just finding another local minimum. Therefore, one should assume that a function is unimodal until proven otherwise. Additionally, one must be careful that artificial local optima, created by numerical noise, are not the reason why one believes the physical

design space is multimodal.

---

**Tip 7.1:** Choose your bounds carefully for gradient-free methods.

Although many gradient-free algorithms are not designed for nonlinear constraints, many do use bound constraints. Unlike gradient-based methods where generous boundaries are often used, for global search methods one must be careful in choosing bounds. Because the optimizer will want to explore throughout the design space, if bounds are unnecessarily wide, the effectiveness of the algorithm will be diminished considerably.

---

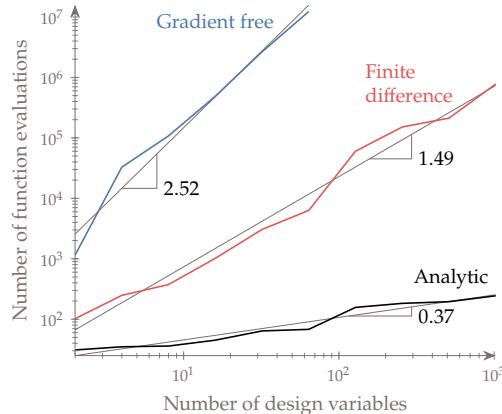
Another reason that is often cited for using a gradient-free method is multiple objectives. Some gradient-free algorithms, like the genetic algorithm discussed in this chapter, can be naturally applied to multiple objectives. However, it is a misconception that gradient-free methods are always preferable just because there are multiple objectives. This topic is discussed in more detail in Chapter 9.

Another common reason for using gradient-free methods is when there are discrete design variables. Since the notion of a derivative with respect to a discrete variable is invalid, gradient-based algorithms cannot be used directly (although there are ways around this limitation as discussed in Chapter 8). Some of the gradient-free algorithms (but not all) can handle discrete variables directly.

The preceding discussion highlights that although multimodality, multiple objectives, or discrete variables, are commonly mentioned as reasons for choosing a gradient-free algorithm, they are not necessarily valid. One of the most relevant factors when choosing between a gradient-free and a gradient-based approach is the dimension of the problem. Figure 7.1 shows how many function evaluations are required to minimize the  $n$ -dimensional Rosenbrock function for varying numbers of design variables. Three classes of algorithms are shown in the plot: gradient-free, gradient-based with finite differenced gradients, and gradient-based with numerically exact gradients. While the exact numbers are problem dependent, similar scaling has been observed on large-scale computational fluid dynamics based optimization <sup>92</sup>. The general takeaway is that for problems of small size (usually  $\leq 30$  variables <sup>93</sup>) gradient-free methods can be useful in finding a solution. Furthermore, because gradient-free methods usually take much less developer time to use, then for these smaller problems a gradient-free solution may even be preferable. However, if the problem is large in dimension then a gradient-based method may be the only feasible path forward despite the need for more developer time.

<sup>92</sup>. Yu et al., *On the Influence of Optimization Algorithm and Starting Design on Wing Aerodynamic Shape Optimization*. 2018

<sup>93</sup>. Rios et al., *Derivative-free optimization: a review of algorithms and comparison of software implementations*. 2013



**Figure 7.1:** Cost of optimization for increasing the number of design variables of the  $n$ -dimensional Rosenbrock function. A gradient-free algorithm compared with a gradient-based algorithm with gradients computed with finite-differences and analytically. A gradient-based optimizer with analytic gradients enables much better scalability.

## 7.2 Classification of Gradient-Free Algorithms

There is a much wider variety of gradient-free algorithms compared to their gradient-based counterparts. While gradient-based algorithms tend to perform local searches, have a mathematical rationale, and be deterministic, gradient-free algorithms exhibit different combinations of these characteristics. We list the most widely known gradient-free algorithms in Table 7.1 and classify them according to the characteristics introduced in Fig. 1.19.\*

**Table 7.1:** Classification of gradient-free optimization methods, using the characteristics of Fig. 1.19.

	Search	Optimal criteria	Iteration proc.	Function eval.	Stochasticity					
	Local	Global	Mathematical	Heuristic	Mathematical	Heuristic	Direct	Surrogate	Deterministic	Stochastic
Nelder–Mead	✓	✓		✓		✓	✓		✓	
GPS	✓		✓		✓		✓		✓	
MADS	✓		✓		✓		✓		✓	
Trust region	✓		✓		✓		✓		✓	
Implicit filtering	✓		✓		✓		✓		✓	
DIRECT		✓	✓		✓		✓		✓	
MCS		✓	✓		✓		✓		✓	
EGO		✓	✓		✓		✓		✓	
SMFs		✓	✓		✓		✓		✓	
Branch and fit		✓		✓		✓	✓		✓	
Hit and run			✓		✓		✓			✓
Evolutionary			✓		✓		✓			✓

\*Rios *et al.*<sup>93</sup> review and benchmark a large selection of gradient-free optimization algorithms.

93. Rios *et al.*, *Derivative-free optimization: a review of algorithms and comparison of software implementations*. 2013

Local-search, gradient-free algorithms that use direct function evaluations include the Nelder–Mead algorithm, generalized pattern search (GPS), and mesh-adaptive direct search (MADS). The Nelder–Mead algorithm (which we detail in Section 7.3) is heuristic, while the other two are not.

GPS and MADS are examples of *derivative-free optimization* (DFO) algorithms, which, in spite of the name, do not include all gradient-free algorithms. DFO algorithms are understood to be largely heuristic-free and focus on local search.<sup>†</sup> GPS is actually a family of methods that iteratively seek an improvement using a set of points around the current point. In its simplest versions, GPS uses a pattern of points based on the coordinate directions, but there are more sophisticated versions that use a more general set of vectors. MADS is an improvement on GPS algorithms by allowing an infinite set of such vectors and improving convergence.<sup>‡</sup>

Model-based, local-search algorithms include trust-region algorithms and implicit filtering. The model is an analytic approximate of the original function (also called a *surrogate model*) and it should be smooth, easy to evaluate, and accurate in the neighborhood of the current point. The trust-region approach detailed in Section 4.5 can be considered gradient-free if the surrogate model is constructed using just evaluations of the original function without evaluating its gradients. This does not prevent the trust-region algorithm from using gradients of the surrogate model, which can be computed analytically. Implicit filtering methods extend the trust region method by adding a surrogate model of the function gradient and use that to guide the search. This effectively becomes a gradient-based method applied to the surrogate model instead of evaluating the function directly as done for the methods in Chapter 4.

Global-search algorithms can be broadly classified as deterministic or stochastic, depending on whether they include random parameter generation within the optimization algorithm.

Deterministic, global-search algorithms can be either direct or model-based. Direct algorithms include Lipschitzian-based partitioning techniques—such as the “divide a hyperrectangle” (DIRECT) algorithm detailed in Section 7.4 and branch and bound search (discussed in Chapter 8)—and multilevel coordinate search (MCS). The DIRECT algorithm selectively divides the space of the design variables into smaller and smaller  $n$ -dimensional boxes (hyperrectangles) and uses mathematical arguments to decide on which boxes should be subdivided.<sup>§</sup> Branch-and-bound search also partitions the design space, but estimates lower and upper bounds for the optimum by using the

<sup>†</sup>The textbooks by Conn *et al.*<sup>94</sup> and Audet *et al.*<sup>95</sup> provide a more extensive treatment of gradient-free optimization algorithms that are based on mathematical criteria.

<sup>94</sup>. Conn *et al.*, *Introduction to Derivative-Free Optimization*. 2009

<sup>95</sup>. Audet *et al.*, *Derivative-Free and Black-box Optimization*. 2017

<sup>‡</sup>The NOMAD software is an open-source implementation of MADS.<sup>96</sup>

<sup>96</sup>. Le Digabel, *Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm*. 2011

<sup>§</sup>DIRECT is one of the few gradient-free methods that has a built-in way to handle constraints that is not a penalty or filtering method.<sup>97</sup>

<sup>97</sup>. Jones, *Direct Global Optimization Algorithm*. 2009

function variation between partitions. MCS is another algorithm that partitions the design space into boxes, where a limit is imposed on how small the boxes can get based on its “level”—the number of times it has been divided.

Model-based global-search algorithms—sometimes called response surface methods (RSMs)—are similar to their local-search algorithm counterparts, but instead of using convex surrogate models, they use surrogate models that can reproduce the features of a multimodal function. One of the most widely used of these algorithms is efficient global optimization (EGO), which uses kriging surrogate models and uses the idea of expected improvement to maximize the likelihood of finding the optimum more efficiently (introduced in Chapter 10). Other algorithms use radial basis functions (RBFs) as the surrogate model and also maximize the probability of improvement at new iterates.

Stochastic algorithms rely on one or more non-deterministic procedures; they include hit and run algorithms, and the broad class of evolutionary algorithms. When performing benchmarks of a stochastic algorithm, you should run a large enough number of optimizations to obtain statistically significant results.

Hit-and-run algorithms generate random steps about the current iterate in search of better points. A new point is accepted when it is better than the current one and this process is repeated until the point cannot be improved.

What constitutes an evolutionary algorithm is not well defined.<sup>¶</sup> Evolutionary algorithms are inspired by processes that occur in nature or society. There is a plethora of evolutionary algorithms in the literature, thanks to the fertile imagination of the research community and a never-ending supply of phenomena for inspiration.<sup>||</sup> These algorithms are more of an analogy of the phenomenon rather than an actual model because they are at best oversimplified models and at worst completely wrong. Nature-inspired algorithms tend to invent their own terminology for the mathematical terms in the optimization problem. For example, a design point might be called a “member of the population,” or the objective function might be referred to as the “fitness.”

The vast majority of evolutionary algorithms are population-based, which means they involve a set of points at each iteration instead of a single one. Because the population is spread out in the design space, evolutionary algorithms perform a global search. The stochastic elements in these algorithms contribute to global exploration and reduce the susceptibility to getting stuck in local minima. These features increase the likelihood of getting close to the global minimum, but by no means guarantee it. The reason it may only get close is that

<sup>¶</sup>Simon<sup>98</sup> provides a more comprehensive review of evolutionary algorithms.

<sup>98</sup>. Simon, *Evolutionary Optimization Algorithms*. 2013

<sup>||</sup> These algorithms include ant colony optimization, artificial bee colony algorithm, design by shopping paradigm, dolphin echolocation algorithm, bacterial foraging optimization, bat algorithm, big bang-big crunch algorithm, biogeography-based optimization, bird mating optimizer, cat swarm optimization, cuckoo search, hybrid glowworm swarm optimization algorithm, mine bomb algorithm, quantum-behaved particle swarm optimization, artificial fish swarm, firefly algorithm, invasive weed optimization, moth-flame optimization algorithm, galactic swarm optimization, hummingbirds optimization algorithm, flower pollination algorithm, artificial flora optimization algorithm, whale optimization algorithm, cockroach swarm optimization, grey wolf optimizer, fruit fly optimization algorithm, imperialist competitive algorithm, harmony search algorithm, penguins search optimization algorithm, intelligent water drops, grenade explosion method, salp swarm algorithm, teaching-learning-based optimization, and water cycle algorithm.

heuristic algorithms have a poor convergence rate, especially in higher dimensions, and because they lack a first-order optimality criterion.

In this chapter, we cover four gradient-free algorithms: the Nelder–Mead algorithm, genetic algorithms, particle swarm optimization, and the DIRECT method. Simulated annealing is covered in Chapter 8 because it was originally developed to solve discrete problems.

### 7.3 Nelder–Mead Algorithm

The simplex method of Nelder *et al.*<sup>21</sup> is a deterministic, direct-search method that is among the most cited of the gradient-free methods. It is also known as the *nonlinear simplex*—not to be confused with the simplex algorithm used for linear programming, with which it has nothing in common.

The Nelder–Mead algorithm is based on a simplex, which is a geometric figure defined by a set of  $n + 1$  points in the design space of  $n$  variables,  $X = \{x^{(0)}, x^{(1)}, \dots, x^{(n)}\}$ . In two dimensions, the simplex is a triangle, and in three dimensions it becomes a tetrahedron. Each optimization iteration is represented by a different simplex. The algorithm consists in modifying the simplex at each iteration using five simple operations. The sequence of operations to be performed is chosen based on the relative values of the objective function at each of the points.

The first step of the simplex algorithm is to generate  $n + 1$  points based on an initial guess for the design variables. This could be done by simply adding steps to each component of the initial point to generate  $n$  new points. However, this will generate a simplex with different edge lengths. Equal length edges are preferable. Suppose we want the length of all sides to be  $l$  and that the first guess is  $x^{(0)}$ . The remaining points of the simplex,  $\{x^{(1)}, \dots, x^{(n)}\}$ , can be computed by

$$x^{(i)} = x^{(0)} + s^{(i)}, \quad (7.1)$$

where  $s^{(i)}$  is a vector whose components  $j$  are defined by

$$s_j^{(i)} = \begin{cases} \frac{l}{n\sqrt{2}} \left( \sqrt{n+1} - 1 \right) + \frac{l}{\sqrt{2}}, & \text{if } j = i \\ \frac{l}{n\sqrt{2}} \left( \sqrt{n+1} - 1 \right), & \text{if } j \neq i \end{cases} \quad (7.2)$$

For example, Fig. 7.2 shows a starting simplex for a two-dimensional problem.

At any given iteration, the objective  $f$  is evaluated for every point, and the points are ordered based on the respective values of  $f$ , from the lowest to the highest. Thus, in the ordered list of simplex points

<sup>21</sup> Nelder *et al.*, *A Simplex Method for Function Minimization*. 1965

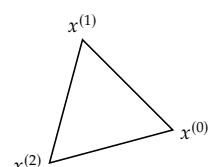


Figure 7.2: Starting simplex for  $n = 2$ .

$X = \{x^{(0)}, x^{(1)}, \dots, x^{(n-1)}, x^{(n)}\}$ , the best point is  $x^{(0)}$ , and the worst one is  $x^{(n)}$ .

The Nelder–Mead algorithm performs four main operations on the simplex to create a new one: *reflection*, *expansion*, *outside contraction*, *inside contraction* and *shrinking*. The operations are shown in Fig. 7.3. Each of these operations, except for the shrinking, generates a new point given by:

$$x = x_c + \alpha (x_c - x^{(n)}) \quad (7.3)$$

where  $\alpha$  is a scalar, and  $x_c$  is the centroid of all the points except for the worst one, i.e.,

$$x_c = \frac{1}{n} \sum_{i=0}^{n-1} x^{(i)}. \quad (7.4)$$

This generates a new point along the line that connects the worst point,  $x^{(n)}$ , and the centroid of the remaining points,  $x_c$ . This direction can be seen as a possible descent direction.

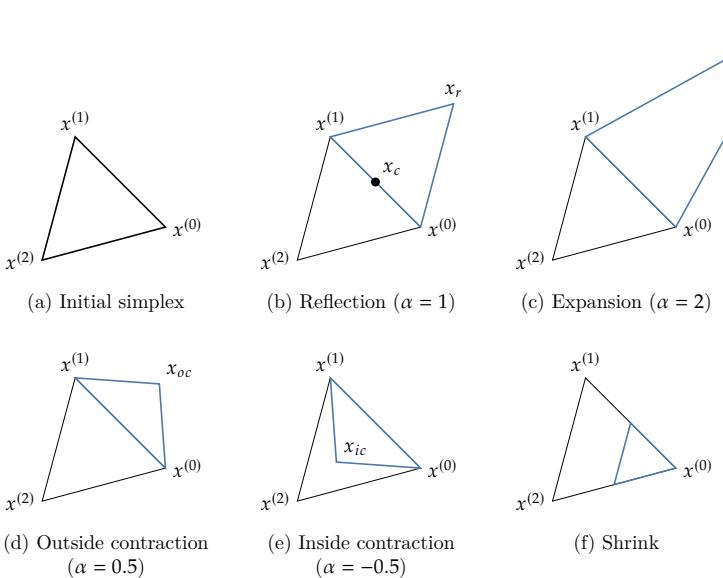


Figure 7.3: Nelder–Mead algorithm operations for  $n = 2$ .

The objective of each iteration is to replace the worst point with a better one to form a new simplex. Each iteration always starts with reflection, which generates a new point using Eq. 7.3 with  $\alpha = 1$  as shown in Fig. 7.3. If the reflected point is better than the best, then the “search direction” was a good one and we go further by performing an expansion using Eq. 7.3 with  $\alpha = 2$ . If the reflected point is between the second worst and the worst, then the direction wasn’t great but it

was at least somewhat of an improvement so we perform an outside contraction ( $\alpha = 1/2$ ). If the reflected point is worse than our worst point we try an inside contraction instead ( $\alpha = -1/2$ ). Shrinking is a last resort operation that is performed when nothing along the line connecting  $x^{(n)}$  and  $x_c$  fails to produce a better point. This operation consists in reducing the size of the simplex by moving all the points closer to the best point,

$$x^{(i)} = x^{(0)} + \gamma \left( x^{(i)} - x^{(0)} \right) \quad \text{for } i = 1, \dots, n, \quad (7.5)$$

where  $\gamma = 0.5$ .

Alg. 7.2 details how a new simplex is obtained for each iteration. In each iteration, the focus is on replacing the worst point with a better one, as opposed to improving the best. The corresponding flowchart is shown in Fig. 7.4.

---

**Algorithm 7.2: Nelder–Mead algorithm**


---

**Inputs:**

$x^{(0)}$ : *Starting point*

$\tau_x$ : *Simplex size tolerances*

$\tau_f$ : *Function value standard deviation tolerances*

**Outputs:**

$x^*$ : *Optimal point*

---

```

for  $j = 1$  to  $n$  do Create a simplex with edge length l
     $x^{(j)} = x^{(0)} + s^{(j)}$   $s^{(j)}$  given by (7.2)
end for

while  $\Delta_x > \tau_x$  or  $\Delta_f > \tau_f$  do Simplex size (7.6) and standard deviation (7.7)
    Sort  $\{x^{(0)}, \dots, x^{(n-1)}, x^{(n)}\}$  Order from the lowest (best) to the highest  $f(x^{(j)})$ 
     $x_c = \frac{1}{n} \sum_{i=0}^{n-1} x^{(i)}$  The centroid excluding the worst point  $x^{(n)}$  (7.4)
     $x_r = x_c + (x_c - x^{(n)})$  Reflection, (7.3) with  $\alpha = 1$ 

    if  $f(x_r) < f(x^{(0)})$  then Is reflected point is better than the best?
         $x_e = x_c + 2(x_c - x^{(n)})$  Expansion, (7.3) with  $\alpha = 2$ 
        if  $f(x_e) < f(x^{(0)})$  then Is expanded point better than the best?
             $x^{(n)} = x_e$  Accept expansion and replace worst point
        else Accept reflection
             $x^{(n)} = x_r$ 
        end if
    else if  $f(x_r) \leq f(x^{(n-1)})$  then Is reflected better than second worst?
         $x^{(n)} = x_r$  Accept reflected point
    else

```

```

if  $f(x_r) > f(x^{(n)})$  then Is reflected point worse than the worst?
     $x_{ic} = x_c - 0.5(x_c - x^{(n)})$  Inside contraction, (7.3) with  $\alpha = -0.5$ 
    if  $f(x_{ic}) < f(x^{(n)})$  then Inside contraction better than worst?
         $x^{(n)} = x_{ic}$  Accept inside contraction
    else
        for  $j = 1$  to  $n$  do
             $x^{(j)} = x^{(0)} + 0.5(x^{(j)} - x^{(0)})$  Shrink, (7.5) with  $\gamma = 0.5$ 
        end for
    end if
else
     $x_{oc} = x_c + 0.5(x_c - x^{(n)})$  Outside contraction, (7.3) with  $\alpha = 0.5$ 
    if  $f(x_{oc}) < f(x_r)$  then Is contraction better than reflection?
         $x^{(n)} = x_{oc}$  Accept outside contraction
    else
        for  $j = 1$  to  $n$  do
             $x^{(j)} = x^{(0)} + 0.5(x^{(j)} - x^{(0)})$  Shrink, (7.5) with  $\gamma = 0.5$ 
        end for
    end if
end if
end while

```

---

The cost for each iteration is one function evaluation if the reflection is accepted, two function evaluations if an expansion or contraction is performed, and  $n + 2$  evaluations if the iteration results in shrinking. Although we could parallelize the  $n$  evaluations when shrinking, it would not be worthwhile because the other operations are sequential.

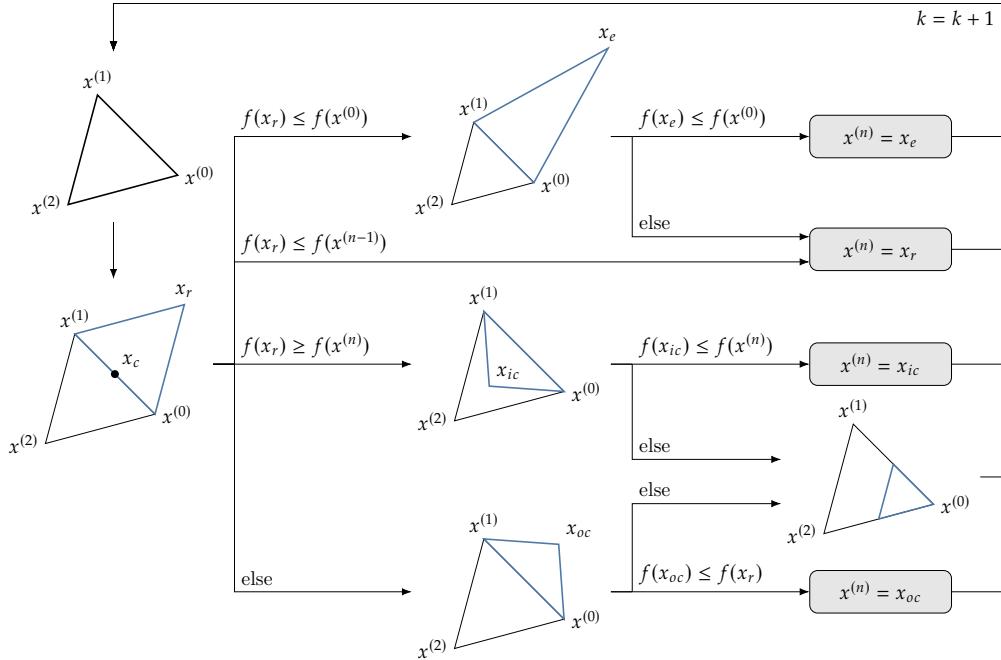
There are a number of ways to quantify the convergence of the simplex method. One straightforward way is to use the size of simplex, i.e.,

$$\Delta_x = \sum_{i=0}^{n-1} \|x^{(i)} - x^{(n)}\|, \quad (7.6)$$

and specify that it must be less than a certain tolerance. Another measure of convergence we can use is the standard deviation the function value,

$$\Delta_f = \sqrt{\frac{\sum_{i=0}^n (f^{(i)} - \bar{f})^2}{n+1}}, \quad (7.7)$$

where  $\bar{f}$  is the mean of the  $n + 1$  function values. Another possible convergence criterion is the difference between the best and worst value in the simplex.



Note that the methodology, like most direct-search methods, cannot directly handle constraints. One approach to handle constraints would be to use a penalty method (discussed in Section 5.3) to form an unconstrained problem. In this case, the penalty does not need to be differentiable, so a linear penalty method would suffice.

Figure 7.4: Flowchart of Nelder–Mead (Alg. 7.2).

**Example 7.3:** Nelder–Mead algorithm applied to the bean function.

Figure 7.5 shows the sequence of simplices that results when minimizing the bean function using a Nelder–Mead simplex. The initial simplex on the upper left is equilateral. The first iteration is a reflection, followed by an inside contraction, another reflection, and an inside contraction before the shrinking. The simplices then shrink dramatically in size, slowly converging to the minimum.

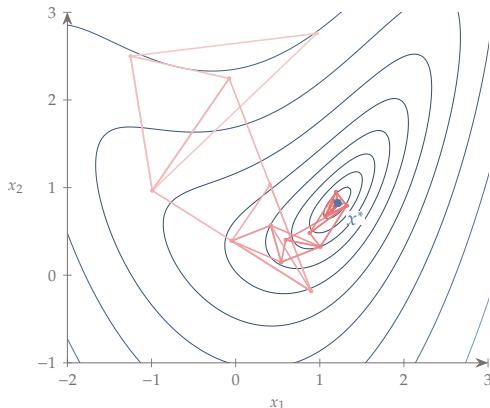
Using a convergence tolerance of  $10^{-6}$  in the difference between  $f_{\text{best}}$  and  $f_{\text{worst}}$  the problem took 68 function evaluations.

## 7.4 DIRECT Algorithm

The divided rectangles (DIRECT) algorithm is different from the other gradient-free optimization algorithms in this chapter in that it is based

<sup>\*\*</sup>This method was developed by Jones *et al.*<sup>45</sup>, who was motivated to develop a global search that did not rely on any tunable parameters (such as population size in genetic algorithms).

<sup>45</sup>Jones *et al.*, Lipschitzian optimization without the Lipschitz constant. 1993



**Figure 7.5:** Sequence of simplices that minimize the bean function

on rigorous mathematics.<sup>\*\*</sup> This is a deterministic method that is guaranteed to converge to the global optimum under conditions that are not too restrictive (although it might require a prohibitive number of function evaluations).

One way to guarantee finding the global optimum within a finite design space is by dividing this space into a regular rectangular grid and evaluating every point in this grid. This is called an *exhaustive search*, and the precision of the minimum depends on how fine the grid is. The cost of this brute-force strategy is high and goes up exponentially with the number of design variables.

The DIRECT method also relies on a grid, but it uses an adaptive meshing scheme that greatly reduces the cost. It starts with a single  $n$ -dimensional hypercube that spans the whole design space—like genetic algorithms, DIRECT requires upper and lower bounds on all the design variables. Each iteration divides this hypercube into smaller ones and evaluates the objective function at the center of each of these. At each iteration, the algorithm only divides rectangles that are determined to be *potentially optimal*. The key strategy in the DIRECT method is how it determines this subset of potentially optimal rectangles, which is based on the mathematical concept of *Lipschitz continuity*.

We start by explaining the concept of Lipschitz continuity and then explain an algorithm for finding the global minimum of a one-dimensional function using this concept—Shubert's algorithm. While Shubert's algorithm is not practical in general, it will help us understand the mathematical rationale for the DIRECT algorithm. Then we explain the DIRECT algorithm for one-dimensional functions before generalizing it for  $n$  dimensions.

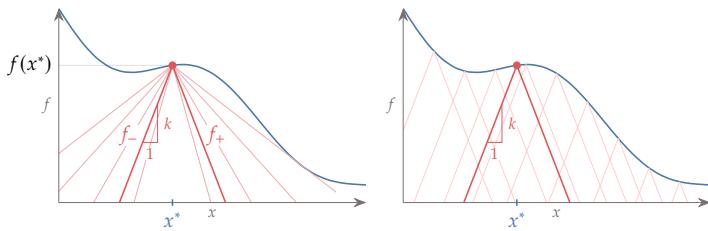
## The Lipschitz Constant

Consider the single variable function  $f(x)$  shown in Fig. 7.6. For a trial point  $x^*$ , we can draw a cone with slope  $k$  by drawing the lines,

$$f_+(x) = f(x^*) + k(x - x^*), \quad (7.8)$$

$$f_-(x) = f(x^*) - k(x - x^*), \quad (7.9)$$

to the left and right, respectively. We show this cone in Fig. 7.6 (left), as well as cones corresponding to other values of  $k$ .



**Figure 7.6:** From a given trial point  $x^*$ , we can draw a cone with slope  $k$  (left). For a function to be Lipschitz continuous, we need all cones with slope  $k$  to lie under the function for all points in the domain (right).

A function  $f$  is said to be *Lipschitz continuous* if there is a cone slope  $k$  such that the cones for all possible trial points in the domain remain under the function. This means that there is a positive constant  $k$  such that

$$|f(x) - f(x^*)| \leq k|x - x^*|, \quad \text{for all } x, x^* \in D, \quad (7.10)$$

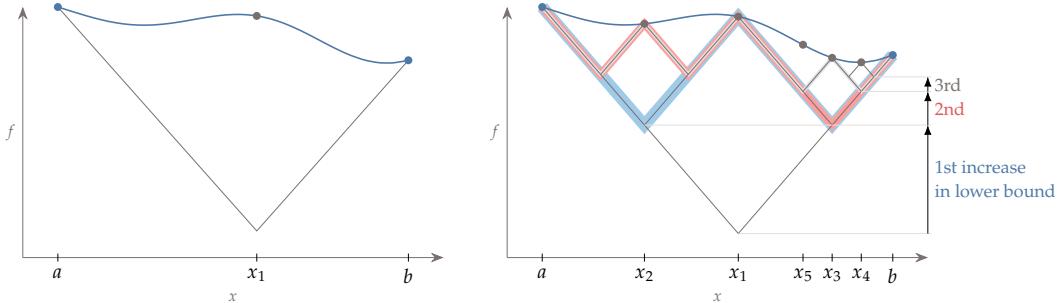
where  $D$  is the function domain. Graphically, this condition means that we can draw a cone with slope  $k$  from any trial point evaluation  $f(x^*)$  such that the function is always bounded by the cone, as shown in Fig. 7.6 (right). Any  $k$  that satisfies condition (7.10) is a *Lipschitz constant* for the corresponding function.

## Shubert's Algorithm

If a Lipschitz constant for a single variable function is known, Shubert's algorithm can find the global minimum of that function. Because the Lipschitz constant is not available in the general case, the DIRECT algorithm is designed so that it does not require this constant. However, we explain Shubert's algorithm first because it provides some of the basic concepts used in the DIRECT algorithm.

Shubert's algorithm starts with a domain within which we want to find the global minimum— $[a, b]$  in Fig. 7.7. Using the property of the Lipschitz constant  $k$  defined in Eq. 7.10, we know that the function is always above a cone of slope  $k$  evaluated at any point in the domain.

We start by establishing a first lower bound on the global minimum by finding the intersection of the cones— $x_1$  in Fig. 7.7 (left)—for the



extremes of the domain. We evaluate the function at  $x_1$  and can now draw a cone about this point to find two more intersections ( $x_2$  and  $x_3$ ). Because these two points always intersect at the same objective lower bound value, they both need to be evaluated to see which one has the highest lower bound increase (the  $x_3$  side in this case). Each subsequent iteration of Shubert's algorithm adds two new points to either side of the current point. These two points are evaluated to find out which side has the lowest actual function value and that side gets selected to be divided.

The lowest bound on the function increases at each iteration and ultimately converges to the *global* minimum. At the same time, the segments in  $x$  decrease in size. The lower bound can switch from distinct regions, as the lower bound in one region increases beyond the lower bound in another region. Using the minimum Lipschitz constant in this algorithm would be the most efficient because it would correspond to the largest possible increments in the lower bound at each iteration.

The two major shortcomings of Shubert's algorithm are that: (1) A Lipschitz constant is usually not available for a general function and (2) it is not easily extended to  $n$  dimension. These two shortcomings are addressed by the DIRECT algorithm.

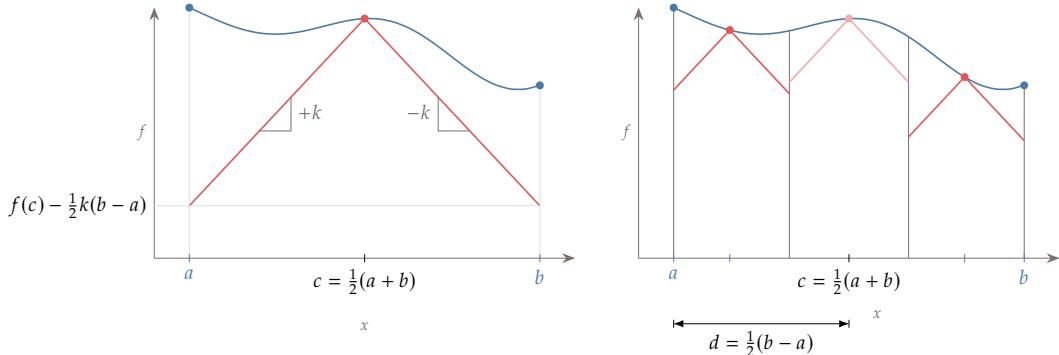
### One-dimensional DIRECT

Before explaining the  $n$ -dimensional DIRECT algorithm, we introduce the one-dimensional version, which is based on principles similar to those of the Shubert algorithm. The main difference is that instead of evaluating at the cone intersection points, we divide the segments evenly and evaluate the center of the segments.

Consider the closed domain  $[a, b]$  shown in Fig. 7.8 (left). For each segment, we evaluate the objective function at the midpoint of the segment. In the first segment, which spans the whole domain, this is  $c_0 = (a + b)/2$ . Assuming some value of  $k$ , which is not known and

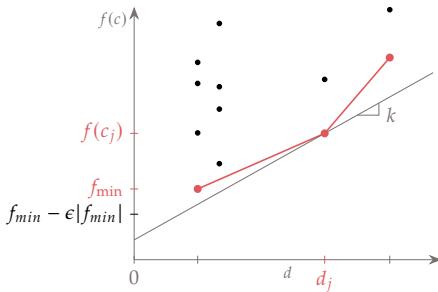
**Figure 7.7:** Shubert's algorithm requires an initial domain and a valid Lipschitz constant (left) and then increases the lower bound of the global minimum with each successive iteration (right).

which we will not need, the lower bound on the minimum would be  $f(c) - k(b - a)/2$ .



We want to increase this lower bound on the function minimum by dividing this segment further. To do this in a regular way that reuses previously evaluated points and can be repeated indefinitely, we divide it into three segments, as shown in Fig. 7.8 (right). Now we have increased the lower bound on the minimum. Unlike the Shubert algorithm, the lower bound is a discontinuous function across the segments, as shown in Fig. 7.8 (right). We now have a regular division of segments, which is more amenable for extending the method to  $n$  dimensions.

Instead of continuing to divide every segment into three other segments, we only divide segments selected according to a *potentially optimal* criterion. To better understand this criterion, consider a set of segments  $[a_i, b_i]$  at a given DIRECT iteration, where segment  $i$  has a half length  $d_i = (b_i - a_i)/2$  and a function value  $f(c_i)$  evaluated at the segment center  $c_i = (a_i + b_i)/2$ . If we plot  $f(c_i)$  versus  $d_i$  for a set of segments, we get the pattern shown in Fig. 7.9.



**Figure 7.8:** The DIRECT algorithm evaluates the middle point (left) and each successive iteration trisects the segments that have the greatest potential (right).

**Figure 7.9:** Potentially optimal segments in the DIRECT algorithm are identified by the lower convex hull of this plot.

The overall rationale for the potentially optimal criterion is that there are two metrics that quantify this potential: the size of the segment and

the function value at the center of the segment. The greater the size of the segment, the greater the potential for containing a minimum. The lower the function value, the greater that potential is as well. For a set of segments of the same size, we know that the one with the lowest function value has the best potential and should be selected. If two segments had the same function value and different sizes, the one with the largest size would should be selected. For a general set of segments with various sizes and value combinations, there might be multiple than can be considered potentially optimal.

We identify potentially optimal segments as follows. If we draw a line with a slope corresponding to a Lipschitz constant  $k$  from any point in Fig. 7.9, the intersection of this line with the vertical axis is a bound on the objective function for the corresponding segment. Therefore, the lowest bound for a given  $k$  can be found by drawing a line through the point that achieves the lowest intersection.

However, we do not know  $k$  and we do not want to assume a value because we do not want to bias the search. If  $k$  were high, it would favor dividing the larger segments. Low values of  $k$  would result in dividing the smaller segments. The DIRECT method hinges on considering all possible values of  $k$ , effectively eliminating the need for this constant.

To eliminate the dependence on  $k$ , we select *all the points for which there is a line with slope  $k$  that does not go above any other point*. This corresponds to selecting the points that form a lower convex hull, as shown by the piecewise linear function in Fig. 7.9. This establishes a lower bound on the function for each segment size.

Mathematically, a segment  $j$  in the set of current segments  $S$  is said to be potentially optimal if there is a  $k \geq 0$  such that

$$f(c_j) - kd_j \leq f(c_i) - kd_i \quad \forall i \in S \quad (7.11)$$

$$f(c_j) - kd_j \leq f_{\min} - \varepsilon |f_{\min}| \quad (7.12)$$

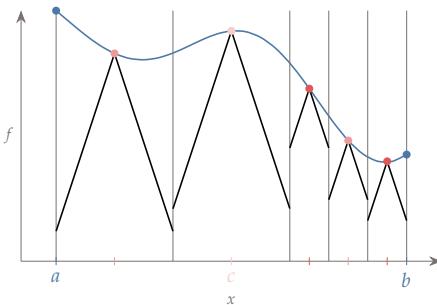
where  $f_{\min}$  is the best current objective function value, and  $\varepsilon$  is a small positive parameter. The first condition corresponds to finding the points in the lower convex hull mentioned previously.

The second condition in Eq. 7.12 ensures that the potential minimum is better than the lowest function value so far by at least a small amount. This prevents the algorithm from becoming too local, wasting function evaluations in search of smaller function improvements. The parameter  $\varepsilon$  balances the search between local and global search. A typical value is  $\varepsilon = 10^{-4}$ , and its the range is usually such that  $10^{-2} \leq \varepsilon \leq 10^{-7}$ .

There are efficient algorithms for finding the convex hull of an arbitrary set of points in two dimensions, such as the Jarvis march.

These algorithms are more than we need here, since we only require the lower part of the convex hull, so they can be simplified for this purpose.

As in the Shubert algorithm, the division might switch from one part of the domain to another, depending on the new function values. When compared to the Shubert algorithm, the DIRECT algorithm produces a discontinuous lower bound on the function values, as shown in Fig. 7.10.



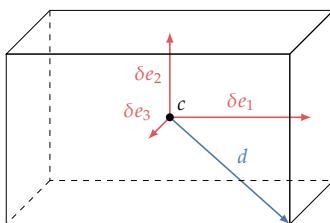
**Figure 7.10:** The lower bound on function values for the DIRECT method are discontinuous at the segment boundaries.

## DIRECT in $n$ Dimensions

The  $n$ -dimensional DIRECT algorithm is similar to the one-dimensional version but becomes more complex.<sup>††</sup> The main differences is that we deal with *hyperrectangles* instead of segments. A hyperrectangle can be defined by its centerpoint position  $c$  in  $n$ -dimensional space and a half length in each direction  $i$ ,  $\delta e_i$ , as shown in Fig. 7.11. The DIRECT algorithm assumes that the initial dimensions are normalized so that we start with a hypercube.

<sup>††</sup>In this chapter, we present an improved version of DIRECT<sup>97</sup>.

97. Jones, *Direct Global Optimization Algorithm*. 2009



**Figure 7.11:** Hyperrectangle in three dimensions, where  $d$  is the maximum distance between the center and the vertices and  $\delta e_i$  is the half-length in each direction  $i$ .

To identify the *potentially optimal rectangles* at a given iteration, we use exactly the same conditions in Eqs. (7.11) and (7.12), but  $c_i$  is now the center of the hyperrectangle, and  $d_i$  is the maximum distance from the center to a vertex. The explanation illustrated in Fig. 7.9 still applies in the  $n$ -dimensional case and still just involves finding the lower convex hull of a set of points with different combinations of  $f$  and  $d$ .

The main complication introduced in the  $n$ -dimensional case is the

division of a selected hyperrectangle. The question is which directions should be divided first. The logic to handle this in the DIRECT algorithm is to prioritize the reduction of the dimensions with the maximum length, which ensures that hyperrectangles do not deviate too much from the proportions of a hypercube. First, we select the set of longest dimensions for intersection (there are often multiple dimensions with the same length). Among this set of longest dimension, we select the direction that has been split the least over the whole history of the search. If there are still multiple dimensions in the selection, we simply select the one with the lowest index. Alg. 7.4 provides the details of this selection and its place in the overall algorithm.

---

**Algorithm 7.4:** DIRECT in  $n$ -dimensions
 

---

**Inputs:** $\bar{x}$ : Variable upper bounds $\underline{x}$ : Variable lower bounds**Outputs:** $x^*$ : Optimal point

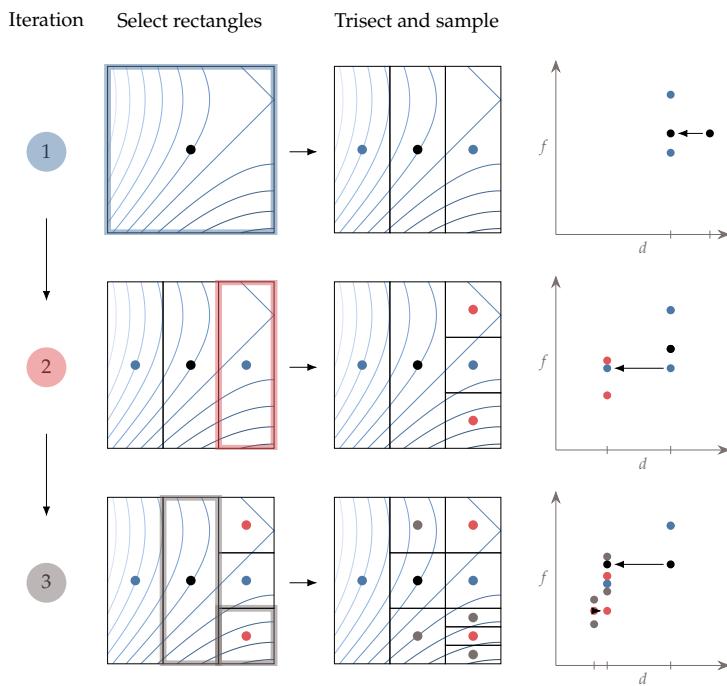
---

Normalize the design space to be the unit hypercube.  
 Compute center of the hypercube,  $c_0$   
 $f_{\min} = f(c_0)$   
**while** not converged **do**  
     Find the set  $S$  of potentially optimal hyperrectangles  
     **for each** hyperrectangle  $r \in S$   
         Find the set  $I$  of dimensions that have maximum side length,  $l_{\max}$ .  
         **if** There is only one maximum side length **then** select it  
         **else** Select the dimension with the lowest number of divisions over the  
             whole history  
             **if** There are more than one selected dimension **then** select the one  
             with the lowest dimension index  
             **end if**  
         **end if**  
     Divide the rectangle into thirds along the selected dimension  
      $k = k + 1$   
**end while**

---

Fig. 7.12 shows the first three iterations for a two-dimensional example and the corresponding visualization of conditions expressed in Eqs. (7.11) and (7.12). We start with a square that contains the whole domain and evaluate the center point. The value of this point is plotted on the  $f$ - $d$  plot on the far right. The first iteration trisects the starting square along the first dimension and evaluates the two new points.

The values for these three points are plotted in the 2nd column from the right in the  $f$ - $d$  plot, where the center point is reused, as indicated by the arrow and the matching color. At this iteration, we have two points that define the convex hull. In the second iteration, we have three rectangles of the same size, so we divide the one with the lowest value and evaluate the centers of the two new rectangles (which are squares in this case). We now have another column of points in the  $f$ - $d$  plot corresponding to a smaller  $d$  and an additional point that defines the lower convex hull. Because the convex hull now has two points, we trisect two different rectangles in the third iteration.



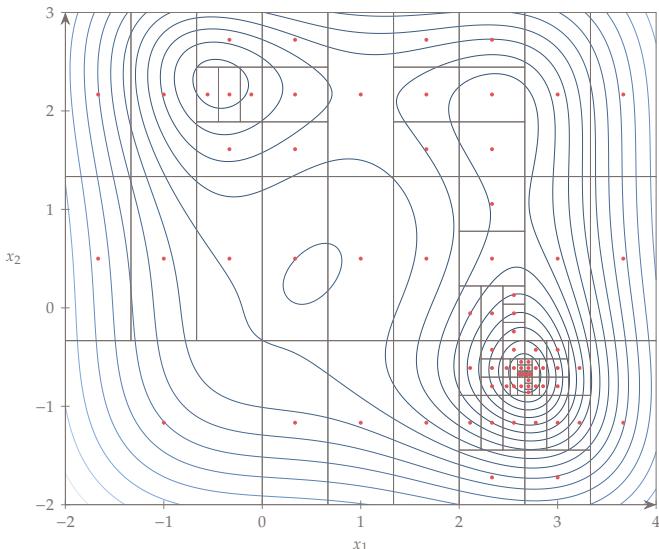
**Figure 7.12:** DIRECT iterations for two-dimensional case (left) and corresponding identification of potentially optimal rectangles (right).

**Example 7.5:** Minimization of multimodal function with DIRECT.

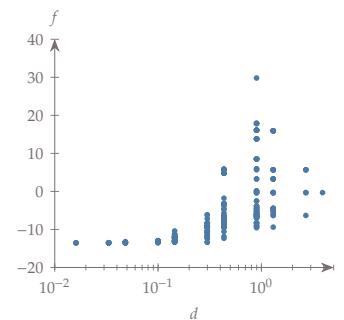
Consider the Jones function, which is a two-dimensional analytic function defined as

$$f(x_1, x_2) = x_1^4 + x_2^4 - 4x_1^3 - 3x_2^3 + 2x_1^2 + 2x_1x_2., \quad (7.13)$$

which has multiple local minima. Applying the DIRECT method to this function, we get the sequence of rectangles shown below.



**Figure 7.13:** CAPTION



**Figure 7.14:** CAPTION

## 7.5 Genetic Algorithms

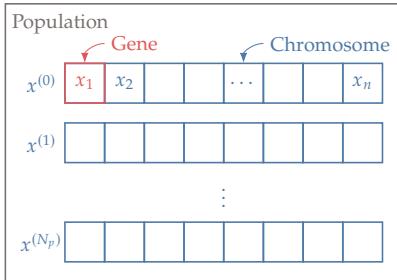
Genetic algorithms (GAs) are the most well-known and widely used type of evolutionary algorithm. They were also among the earliest to have been developed.<sup>#</sup> GAs, like many evolutionary algorithms, are *population based*: The optimization starts with a set of design points (the population) rather than a single starting point, and each optimization iteration updates this set in some way. Each iteration in the GA is called a *generation*, and each generation has a population with  $N_p$  points. A *chromosome* is used to represent each point and contains the values for all the design variables, as shown in Fig. 7.15. Each design variable is represented by a *gene*. As we will see later, there are different ways for genes to represent the design variables.

GAs evolve the population using an algorithm inspired by biological reproduction and evolution using three main steps: 1) selection, 2)

<sup>#</sup>The first GA software was written in 1954, followed by other seminal work.<sup>99</sup> Initially, these GAs were not written to perform optimization, but rather, to model the evolutionary process. GAs were eventually applied to optimization.<sup>100</sup>

<sup>99</sup>. Barricelli, *Esempi numerici di processi di evoluzione*. 1954

<sup>100</sup>. Jong, *An analysis of the behavior of a class of genetic adaptive systems*. 1975



**Figure 7.15:** Each GA iteration involves a population of design points where each design is represented by a chromosome and each design variable is represented by a gene.

crossover , and 3) mutation. Selection is based natural selection, where members of the population that acquire favorable adaptations survive longer and contribute more to the population gene pool. Crossover is inspired by chromosomal crossover, which is the exchange of genetic material between chromosomes during sexual reproduction. In this step, two parents produce two offspring. Mutation mimics genetic mutation, which is a permanent change in the gene sequence that occurs naturally.

Alg. 7.6 and Fig. 7.16 show how these three steps are used to perform optimization. Although most GAs follow this general procedure, there is a great degree of flexibility in how the steps are performed, leading to many variations in GAs. For example, there is no one single method specified for the generation of the initial population, and the size of that population varies. Similarly, there are many possible methods for selecting the parents, for generating the offspring, and for selecting the survivors. Here, the new population ( $P_{k+1}$ ) is formed exclusively by the offspring generated from crossover. However, some GAs add an extra selection process that selects a surviving population of size  $N_p$  among the population of parents and offspring.

---

#### Algorithm 7.6: Genetic algorithm

---

##### Inputs:

$\bar{x}$ : Variable upper bounds

$\underline{x}$ : Variable lower bounds

##### Outputs:

$x^*$ : Optimal point

---

$$k = 0$$

$$P_k = \{x^{(1)}, x^{(2)}, \dots, x^{(N_p)}\}$$

**while**  $k < k_{\max}$  **do**

    Compute  $f(x) \forall x \in P_k$

    Select  $N_p/2$  parent pairs from  $P_k$  for crossover

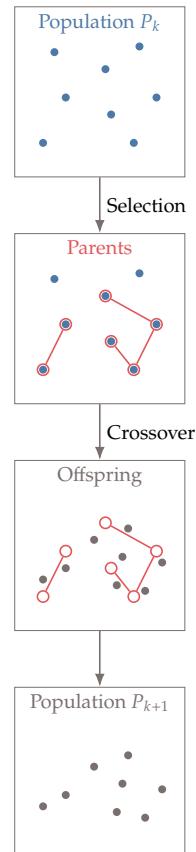
    Generate a new population of  $N_p$  offspring ( $P_{k+1}$ )

*Generate initial population*

*Evaluate fitness*

*Selection*

*Crossover*



**Figure 7.16:** At each GA iteration, pairs of parents are selected from the population to generate the offspring through crossover, which become the new population.

---

```

Randomly mutate some points in the population           Mutation
k = k + 1
end while

```

---

In addition to the flexibility in the various operations, there are also different methods for representing the design variables in a genetic algorithm. The design variable representation can be used to classify genetic algorithms into two broad categories: *binary-encoded* and *real-encoded* genetic algorithms. Binary-encoded algorithms use bits to represent the design variables, while the real-encoded algorithms keep the same real value representation used in most other algorithms. The details of the operations in Alg. 7.6 depend on whether we are using one or the other of these representations, but the principles remain the same. In the rest of this section, we describe in more detail a particular way of performing these operations for each of the possible design variable representations.

### 7.5.1 Binary-encoded Genetic Algorithms

The original genetic algorithms were based on binary encoding because they more naturally mimic chromosome encoding. Binary-coded GAs are widely used and are applicable to discrete or mixed-integer problems.<sup>§§</sup> When using binary encoding, we represent each variable as a binary number with  $m$  bits. Each bit in the binary representation has a *location*,  $i$ , and a *value*,  $b_i$  (which is either 0 or 1). If we want to represent a real-valued variable, we first need to consider a finite interval  $x \in [\underline{x}, \bar{x}]$ , which we can then divide into  $2^m - 1$  intervals. The size of the interval is given by

$$\Delta x = \frac{\bar{x} - \underline{x}}{2^m - 1}. \quad (7.14)$$

To have a more precise representation, we must use more bits.

When using binary-encoded GAs, we do not need to encode the design variables (since they are generated and manipulated directly in the binary representation), but we do need to decode them before providing them to the evaluation function. To decode a binary representation, we use

$$x = \underline{x} + \sum_{i=0}^{m-1} b_i 2^i \Delta x. \quad (7.15)$$

<sup>§§</sup>One popular binary-encoded genetic algorithm implementation is NSGA-II <sup>101</sup>.

<sup>101</sup>. Deb et al., *A fast and elitist multiobjective genetic algorithm: NSGA-II*. 2002

---

**Example 7.7:** Binary representation of a real number.

Suppose we have a continuous design variable  $x$  that we want to represent in the interval  $[-20, 80]$  using 12 bits. Then, we have  $2^{12-1} = 4,095$  intervals, and using Eq. 7.14, we get  $\Delta x \approx 0.0244$ . This interval is the error in this finite precision representation. For the sample binary representation shown below, we can use Eq. 7.15 to compute the equivalent real number, which turns out to be  $x \approx 32.55$ .

$i$	1	2	3	4	5	6	7	8	9	10	11	12
$b_i$	0	0	0	1	0	1	1	0	0	0	0	1

## Initial Population

The first step in a genetic algorithm is to generate an initial set (population) of points. As a rule of thumb, the population size should be approximately one order of magnitude larger than the number of design variables, but in general you will need to experiment with different population sizes.

One popular way to choose the initial population is to do it at random. Using binary encoding, we can assign each bit in the representation of the design variables a 50% chance of being either 1 or 0. This can be done by generating a random number  $0 \leq r \leq 1$  and setting the bit to 0 if  $r \leq 0.5$  and 1 if  $r > 0.5$ . For a population of size  $N_p$ , with  $n_x$  design variables, and each variable is encoded using  $m$  bits, the total number of bits that needs to be generated is  $N_p \times n_x \times m$ .

To achieve better spread in a larger dimension space, methods like Latin hypercube sampling are generally more effective than random populations (discussed in Section 10.2).

## Evaluate Fitness

The objective function for all the points in the population must be evaluated and then converted to a fitness value. These evaluations could be done in parallel. The numerical optimization convention is usually to minimize the objective, while the GA convention is to maximize the fitness. Therefore, we can convert the objective to fitness simply by setting  $F = -f$ .

For some types of selection (like the tournament selection detailed in the next step) all the fitness values need to be positive. To achieve that, we can perform the following conversion:

$$F = \frac{-f_i + \Delta F}{\max(1, \Delta F - f_{\text{low}})} , \quad (7.16)$$

where  $\Delta F = 1.1f_{\text{high}} - 0.1f_{\text{low}}$  is based on the highest and lowest function values in the population, and the denominator is introduced to scale the fitness.

## Selection

In this step we choose points from the population for reproduction in a subsequent step. On average, it is desirable to choose a mating pool that improves in fitness (thus mimicking the concept of natural selection), but it is also important to maintain diversity. In total, we need to generate  $N_p/2$  pairs.

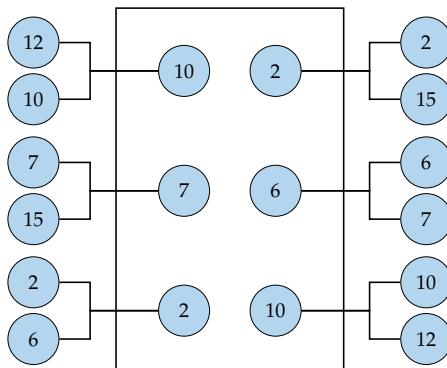
The simplest selection method is to randomly select two points from the population until the requisite number of pairs is complete. This approach is not particularly effective because there is no mechanism to move the population toward points with better objective functions.

*Tournament selection* is a better method that randomly pairs up  $N_p$  points, and selects the best point from each pair to join the *mating pool*. The same pairing and selection process is repeated to create  $N_p/2$  more points to complete a mating pool of  $N_p$  points.

---

**Example 7.8:** Tournament selection process.

Figure 7.17 illustrates the process with a very small population. Each member of the population ends up in the mating pool zero, one, or two times with better points more likely to appear in the pool. The best point in the population will always end up in the pool twice, while the worst point in the population will always be eliminated.



**Figure 7.17:** Tournament selection example.

---

Another common method is *roulette wheel selection*. This concept is patterned after a roulette wheel used in a casino. It assigns better

points a larger sector on the roulette wheel so that they have a higher probability of being selected.

To find the sizes of the sectors in the roulette wheel selection, we use the fitness value defined by Eq. 7.16. We then take the normalized cumulative sum of the scaled fitness values to compute an interval for each members in the population  $j$  as

$$S_j = \frac{\sum_{i=1}^j F_i}{\sum_{i=1}^{N_p} F_i} \quad (7.17)$$

We can now create a mating pool of  $N_p$  points by turning the roulette wheel  $N_p$  times. We do this by generating a random number  $0 \leq r \leq 1$  at each turn. The  $j^{\text{th}}$  member is copied to the mating pool if

$$S_{j-1} < r \leq S_j \quad (7.18)$$

This ensures that the probability of a member being selected for reproduction is proportional to its scaled fitness value.

---

**Example 7.9:** Roulette wheel selection process.
 

---

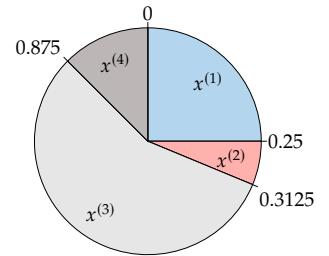
Assume that  $F = [20, 5, 45, 10]$ . Then  $S = [0.25, 0.3125, 0.875, 1]$ , which divides the “wheel” into four segments shown graphically as show in Fig. 7.18.

## Crossover

In the reproduction operation, two points (offspring) are generated from a pair of points (parents). Various strategies are possible in genetic algorithms. *Single-point crossover* usually involves generating a random integer  $1 \leq k \leq m - 1$  that defines the *crossover point*. This is illustrated in Table 7.2. For one of the offspring, the first  $k$  bits are taken from, say, parent 1 and the remaining bits from parent 2. For the second offspring, the first  $k$  bits are taken from parent 2 and the remaining ones from parent 1. Various extensions exist like two-point crossover or  $n$ -point crossover.

## Mutation

Mutation is a random operation performed to change the genetic information and is needed because even though selection and reproduction effectively recombine existing information, occasionally some useful



**Figure 7.18:** Roulette wheel selection example.

**Table 7.2:** Single-point crossover operation example.

Before crossover	After crossover
11 111	11 000
00 000	00 111

genetic information might be lost. The mutation operation protects against such irrecoverable loss and introduces additional diversity into the population.

When using bit representation, every bit is assigned a small permutation probability, say  $p = 0.005 \sim 0.1$ . This is done by generating a random number  $0 \leq r \leq 1$  for each bit, which is changed if  $r < p$ . An example is illustrated in Table 7.3.

**Table 7.3:** Mutation example where only one bit changed.

Before mutation	After mutation
11111	11011

## 7.5.2 Real-encoded Genetic Algorithms

As the name implies, real-encoded GAs represent the design variables in their original representation as real numbers. This has several advantages over the binary-encoded approach. First, real-encoding represents numbers up to machine precision rather than being limited by the initial choice of string length required in binary-encoding. Second, it avoids the “Hamming cliff” issue of binary-encoding, which is caused by the fact that a large number of bits must change to move between adjacent real numbers (e.g., 0111 to 1000). Third, some real-encoded GAs are able to generate points outside the design variable bounds used to create the initial population; in many problems, the design variables are not bounded. Finally, it avoids the burden of binary coding and decoding. The main disadvantage is that integer or discrete variables cannot be handled in a straightforward way. For problems that are continuous, a real-encoded GA is generally more efficient than a binary-encoded GA <sup>98</sup>. We now describe the required changes to the GA operations in the real-encoded approach.

<sup>98.</sup> Simon, *Evolutionary Optimization Algorithms*. 2013

### Initial Population

The most common approach is to pick the  $N_P$  points using random sampling within the provided design bounds. Each member is often

chosen at random within some initial bounds. For each design variable  $x_i$ , with bounds such that  $\underline{x}_i \leq x_i \leq \bar{x}_i$ , we could use,

$$x_i = \underline{x}_i + r(\bar{x}_i - \underline{x}_i) \quad (7.19)$$

where  $r$  is a random number such that  $0 \leq r \leq 1$ .

Again, for higher dimensional spaces Latin hypercube sampling can provide better coverage.

## Selection

The selection operation does not depend on the design variable encoding, and therefore, we can just use any of the selection approaches already described in the binary-encoded GA.

## Crossover

When using real-encoding, the term “crossover” does not accurately describe the process of creating the two offspring from a pair of points. Instead, the approaches are more accurately described as a *blending*, although the name crossover is still often used.

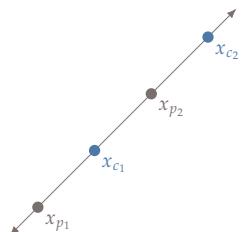
There are various options for the reproduction of two points encoded using real numbers. A common method is *linear crossover*, which generates two or more points in the line defined by the two parent points. One option for linear crossover is to generate the following two points:

$$\begin{aligned} x_{c_1} &= 0.5x_{p_1} + 0.5x_{p_2}, \\ x_{c_2} &= 2x_{p_2} - x_{p_1}, \end{aligned} \quad (7.20)$$

where parent two is more fit than parent one ( $f(x_{p_2}) < f(x_{p_1})$ ). An example of this linear crossover approach is shown in Fig. 7.19, where we can see that child 1 is the average of the two parent points, while child 2 is obtained by extrapolating in the direction of the “fitter” parent.

Another option is a simple crossover like the binary case where a random integer is generated to split the vectors. For example with a split after the first index:

$$\begin{aligned} x_{p_1} &= [x_1, x_2, x_3, x_4] \\ x_{p_2} &= [x_5, x_6, x_7, x_8] \\ &\Downarrow \\ x_{c_1} &= [x_1, x_6, x_7, x_8] \\ x_{c_2} &= [x_5, x_2, x_3, x_4] \end{aligned} \quad (7.21)$$



**Figure 7.19:** Linear crossover produces two new points along the line defined by the two parent points.

This simple crossover does not generate as much diversity as the binary case does and relies more heavily on effective mutation. Many other strategies have been devised for real-encoded GAs<sup>102</sup>.

102. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. 2001

## Mutation

Like a binary-encoded GA, mutation should only occur with a small probability (e.g.,  $p = 0.005 \sim 0.1$ ). However, rather than changing each bit with probability  $p$ , we now change each design variable with probability  $p$ .

Many mutation methods rely on random variations around an existing member such as a uniform random operator:

$$x_{\text{new}i} = x_i + (r_i - 0.5)\Delta_i, \text{ for } i = 1 \dots n_x \quad (7.22)$$

where  $r_i$  is a random number between 0 and 1, and  $\Delta_i$  is a pre-selected maximum perturbation in the  $i^{\text{th}}$  direction. Many non-uniform methods exist as well. For example, we can use a Gaussian distribution

$$x_{\text{new}i} = x_i + \mathcal{N}(0, \sigma_i), \text{ for } i = 1 \dots n_x \quad (7.23)$$

where  $\sigma_i$  is a pre-selected standard deviation and random samples are drawn from the normal distribution. During the mutation operations, bound checking is necessary to ensure the mutations stay within the upper and lower limits.

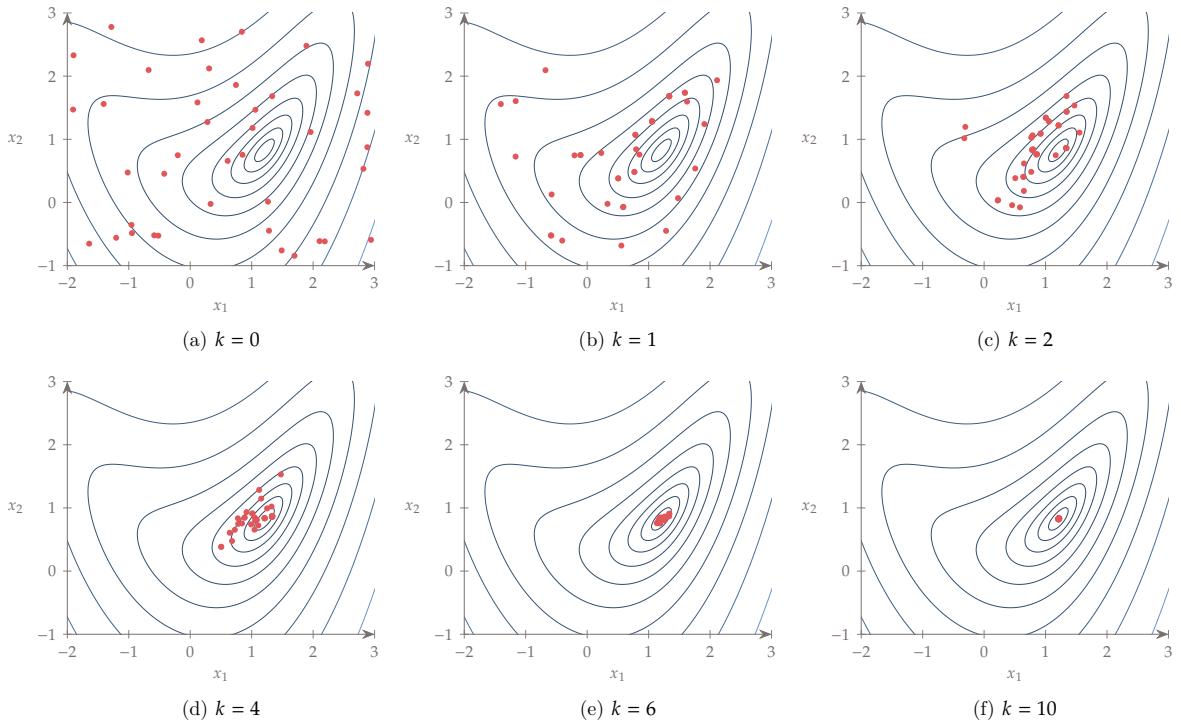
**Example 7.10:** Genetic algorithm applied to the bean function.

Figure 7.20 shows the evolution of the population when minimizing the bean function using a genetic algorithm. The initial population size was 40, and the simulation was run for 14 generations, requiring 2000 total function evaluations. Convergence was assumed if the best member in the population improved by less than  $10^{-4}$  for 3 consecutive generations.

### 7.5.3 Constraint Handling

Various approaches exist for handling constraints. Like the Nelder-Mead method, we can use a penalty method (e.g., Augmented Lagrangian, linear penalty, etc.). However, there are additional options for GAs. In the tournament selection, we can use other selection criteria that do not depend on penalty parameters. One such approach for choosing the best selection amongst two competitors is:

1. prefer a feasible solution



2. among two feasible solutions, choose the one with a better objective
3. among two infeasible solutions, choose the one with a smaller constraint violation

This concept is a lot like the filter methods discussed in Section 5.6.

#### 7.5.4 Convergence

Rigorous mathematical convergence criteria, like those used in gradient-based optimization, do not apply to genetic algorithms. The most common way to terminate a genetic algorithm is to simply specify a maximum number of iterations, which corresponds to a computational budget. Another similar approach is to run indefinitely until the user manually terminates the algorithm, usually by monitoring the trends in population fitness.

A more automated approach is to track a running average of the population fitness, although it can be difficult to decide what tolerance to apply to this criterium as we generally aren't interested in the average performance anyway. Perhaps a more direct metric of interest is to track the fitness of the best member in the population. However, this

**Figure 7.20:** Population evolution at iterations  $k$  using a genetic algorithm to minimize the bean function

can be a problematic criterium to use because the best member can disappear due to crossover or mutation. To avoid this, and to improve convergence, many genetic algorithms employ *elitism*. This means that the fittest member in the population is retained so that the population is guaranteed to never regress. Even without this behavior, the best member often changes slowly, so one should not terminate unless the best member has not improved for several generations.

## 7.6 Particle Swarm Optimization

Like a GA, particle swarm optimization (PSO) is a stochastic population-based optimization algorithm based on the concept of “swarm intelligence”. Swarm intelligence is the property of a system whereby the collective behaviors of unsophisticated agents interacting locally with their environment cause coherent global patterns to emerge. In other words: dumb agents, properly connected into a swarm, can yield smart results.<sup>103</sup>

The “swarm” in PSO is a set of design points (“agents” or “particles”) that move in  $n$ -dimensional space looking for the best solution. Although these are just design points, the history for each point is relevant to the PSO algorithm, so we use adopt the term “particle”. Each particle moves according to a velocity, and this velocity changes according to the past objective function values of that particle and the current objective values of the rest of the particles. Each particle remembers the location where it found its best result so far and it exchanges information with the swarm about the location where the swarm has found the best result so far.

The position of particle  $i$  for iteration  $k + 1$  is updated according to

$$x_{k+1}^{(i)} = x_k^{(i)} + v_{k+1}^{(i)} \Delta t, \quad (7.24)$$

where  $\Delta t$  is a constant artificial time step. The velocity for each particle is updated as follows:

$$v_{k+1}^{(i)} = \bar{w} v_k^{(i)} + c_1 r_1 \frac{x_{\text{best}}^{(i)} - x_k^{(i)}}{\Delta t} + c_2 r_2 \frac{x_{\text{best}} - x_k^{(i)}}{\Delta t}. \quad (7.25)$$

The first component in this update is the “inertia”, which, through the parameter  $\bar{w}$ , dictates how much the new velocity should tend to be the same as the one in the previous iteration.

The second term represents “memory” and is a vector pointing toward the best position particle  $i$  has seen in all its iterations so far,  $x_{\text{best}}^{(i)}$ . The weight in this term consists of a constant  $c_1$ , and a random

<sup>103</sup>PSO was first proposed by Eberhart and Kennedy <sup>103</sup>. Eberhart was an electrical engineer and Kennedy was a social-psychologist.

<sup>103</sup>. Eberhart *et al.*, *New Optimizer Using Particle Swarm Theory*. 1995

parameter  $r_1$  in the interval  $[0, 1]$  that introduces a stochastic component to the algorithm. Thus,  $c_1$  controls how much of an influence the best point found by the particle so far has on the next direction.

The third term represents “social” influence. It behaves similarly to the memory component, except that  $x_{\text{best}}$  is the best point the entire swarm has found so far, and  $c_2$  controls how much of an influence this best point has in the next direction. The relative values of  $c_1$  and  $c_2$  thus control the tendency toward local versus global search, respectively.

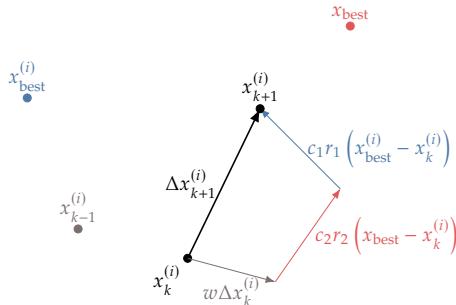
Since the time step is artificial, we can eliminate it by multiplying Eq. 7.25 by  $\Delta t$  to yield a step

$$\Delta x_{k+1}^{(i)} = w \Delta x_k^{(i)} + c_1 r_1 (x_{\text{best}}^{(i)} - x_k^{(i)}) + c_2 r_2 (x_{\text{best}} - x_k^{(i)}). \quad (7.26)$$

We then use this step to update the particle position for the next iteration, i.e.,

$$x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)}. \quad (7.27)$$

The three components of the update (7.26) are shown in Fig. 7.21 for a two-dimensional case.



**Figure 7.21:** Components of the PSO update.

Typical values for the inertia parameter  $w$  are in the interval  $[0.8, 1.2]$ . A lower value of  $w$  reduces the particle’s inertia and tends toward faster convergence to a minimum, while a higher value of  $w$  increases the particle’s inertia and tends toward increased exploration to potentially help discover multiple minima. Thus, there is a tradeoff in this value. Both  $c_1$  and  $c_2$  values are in the interval  $[0, 2]$ , and typically closer to 2.

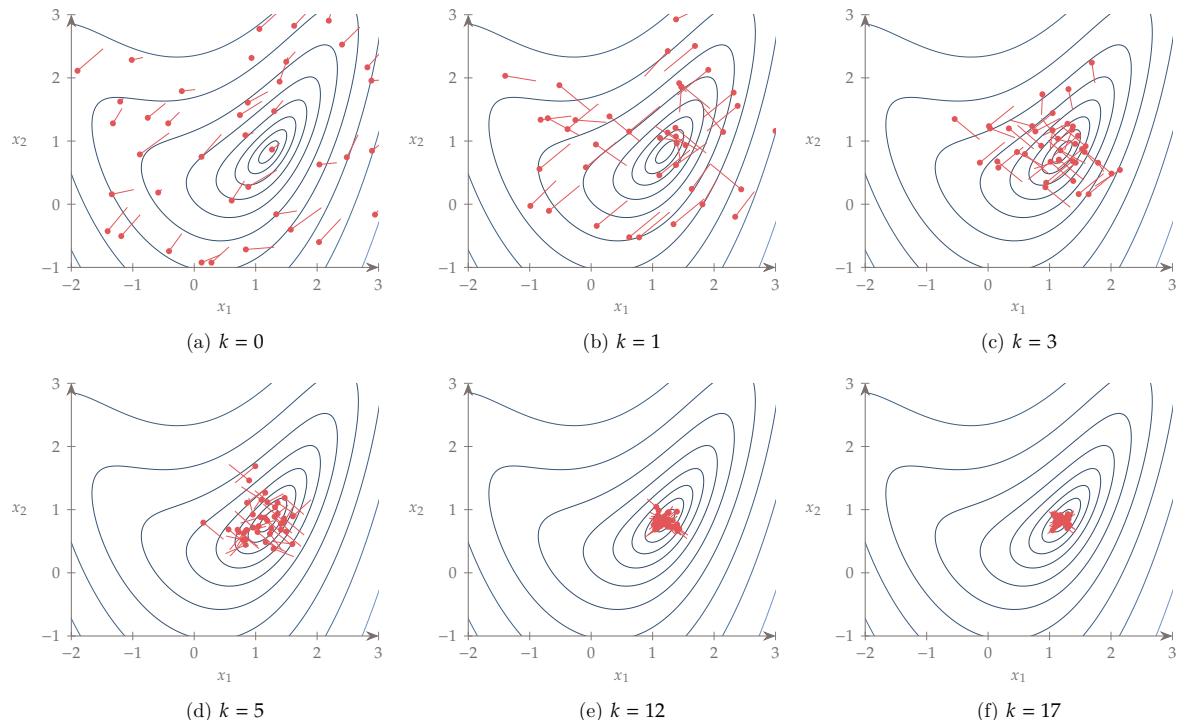
The first step in the PSO algorithm is to initialize the set of particles (Alg. 7.12). Like a GA, the initial set of points can be determined at random or can use a more sophisticated design of experiments strategy (like Latin hypercube sampling). The main loop in the algorithm computes the steps to be added to each particle and updates their positions. A number of convergence criteria are possible, some of which are similar to the simplex method and GA: the distance (sum

or norm) between each particle and the best particle falls below some tolerance, the best particle's fitness changes by less than some tolerance across multiple generations, the difference between the best and worst member falls below some tolerance. In the case of PSO, another alternative is to check whether the velocities for all particles (norm, mean, etc.) falls below some tolerance. Some of these criteria that assume all the particles will congregate (distance, velocities) don't work well for multimodal problems. In those cases tracking just the best particle's fitness may be more desirable.

---

**Example 7.11:** PSO algorithm applied to the bean function.

Figure 7.22 shows the sequence of simplices that results when minimizing the bean function using a particle swarm method. The initial population size was 40 and the optimization required 600 function evaluations. Convergence was assumed if the best value found by the population did not improve by more than  $10^{-4}$  for 3 consecutive iterations.



**Figure 7.22:** Sequence of particles at iterations  $k$  that minimize the bean function

---

**Algorithm 7.12:** Particle swarm optimization algorithm

**Inputs:**

$\bar{x}$ : Variable upper bounds  
 $\underline{x}$ : Variable lower bounds  
 $w$ : "Inertia" parameter  
 $c_1$ : Self influence parameter  
 $c_2$ : Social influence parameter

**Outputs:**

$x^*$ : Optimal point

```

 $k = 0$ 
for all i do Loop to initialize all particles
    Generate position  $x_0^{(i)}$  within specified bounds.
     $x_{\text{best}}^{(i)} = x_0^{(i)}$  First position is the best so far.
    Evaluate  $f(x_0^{(i)})$ 
    if  $i = 0$  then
         $x_{\text{best}} = x_0^{(i)}$ 
    else
        if  $f(x_0^{(i)}) < f(x_{\text{best}})$  then
             $x_{\text{best}} = x_0^{(i)}$ 
        end if
    end if
    Initialize "velocity"  $\Delta x_k^{(i)}$ 
end for
while not converged do Main iteration loop
     $\Delta x_{k+1}^{(i)} = w\Delta x_k^{(i)} + c_1 r_1 \left( x_{\text{best}}^{(i)} - x_k^{(i)} \right) + c_2 r_2 \left( x_{\text{best}} - x_k^{(i)} \right)$ 
     $x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)}$  Update the particle position while enforcing bounds.
    if  $x_{k+1}^{(i)} < x_{\text{lower}}$  or  $x_{k+1}^{(i)} > x_{\text{upper}}$  then
         $\Delta x_{k+1}^{(i)} = c_1 r_1 \left( x_{\text{best}}^{(i)} - x_k^{(i)} \right) + c_2 r_2 \left( x_{\text{best}} - x_k^{(i)} \right)$ 
         $x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)}$ 
    end if
    for all  $x_{k+1}^{(i)}$  do
        Evaluate  $f(x_{k+1}^{(i)})$ 
        if  $f(x_{k+1}^{(i)}) < f(x_{\text{best}}^{(i)})$  then
             $x_{\text{best}}^{(i)} = x_{k+1}^{(i)}$ 
        end if
    end for
     $k = k + 1$ 
end while

```

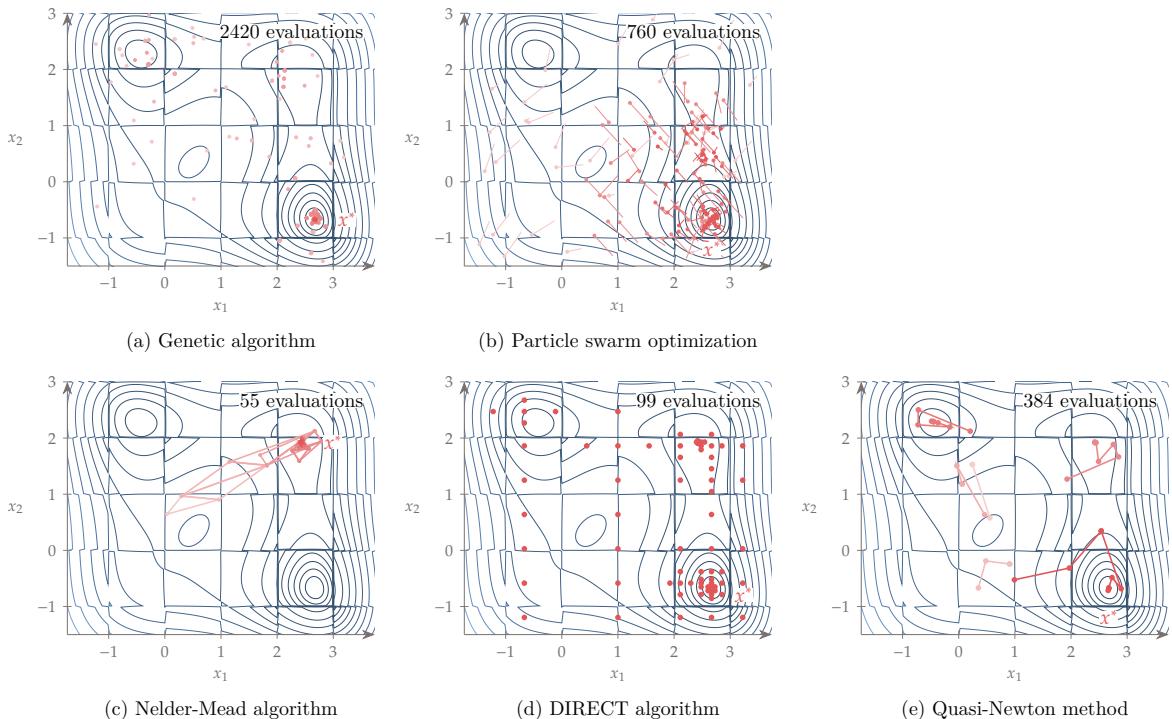
---

Example 7.13: Comparison of algorithms for multimodal function

We now return to the Jones function (Eq. 7.13 used in Ex. 7.5 to demonstrate the DIRECT method), but make it discontinuous by adding the following function:

$$\Delta f = 4 \lceil \sin(\pi x_1) \sin(\pi x_2) \rceil. \quad (7.28)$$

By taking the ceiling of the product of the two sine waves, this function creates a checkerboard pattern with zeros and fours. Adding this function to the Jones function produces the discontinuous function shown in Fig. 7.23, where we can clearly see the discontinuities. The global optimum remains the same as the original function. The resulting optimization paths demonstrate that the gradient-free algorithms are effective. Both the GA and PSO find the global minimum, but they require a large number of evaluations for the same accuracy. Nelder–Mead converges quickly, but not to the global minimum.



**Figure 7.23:** Convergence path for gradient-free algorithms compared to gradient-based with multistart.

**Tip 7.14:** Compare optimization algorithms fairly.

It is difficult to make a fair comparison between different algorithms, especially when they use different convergence criteria. You can either compare the computational cost of achieving an objective with a specified accuracy, or compare the objective achieved for a specified computational cost. To compare algorithms that use different convergence criteria, you can run them for as long

as you can afford with the lowest convergest tolerance possible and tabulate the number of function evaluations and the respective objective function values. To compare the computational cost for a specified tolerance, you can determine the number of function evaluations that each algorithm requires to achieve a given number of digits agreement in the objective function. Alternatively, you can compare the objective achieved for the different algorithms for a given number of function evaluations. Comparison becomes more challenging for constrained problems because a better objective that is less feasible is not necessarily better. In that case, you need to make sure that all results are feasible to the same tolerance. When comparing algorithms that include stochastic procedures (e.g., GA, PSO), you should run each optimizatin multiple times to get statistically significant data and compare the mean and variance of the performance metrics.

---

## 7.7 Summary

Gradient-free optimization algorithms are needed when the objective and constraint functions are not smooth enough or when it is not possible to compute derivatives with enough precision. One major advantage of gradient-free methods is that they tend to be robust to numerical noise and discontinuities, which makes them easier to use than gradient-based methods.

However, the overall cost of gradient-free optimization is sensitive to the cost of the function evaluations because the number of iterations required for convergence scales poorly with the number of design variables.

There is a wide variety of gradient-free methods. They can perform local or global search, use mathematical or heuristic criteria, and be deterministic or stochastic. A global search does not guarantee convergence to the global optimum, but increases the likelihood of such convergence. We should be wary when heuristics are used to establish convergence because the result might not correspond to the true mathematical optimum. Heuristics in the optimization algorithm also limit the rate of convergence compared to algorithms based on mathematical principles.

Evolutionary algorithms are global search methods based on the evolution of a population of designs. They are based on heuristics inspired by natural or societal phenomena and have some stochastic element in their algorithm. The genetic algorithm (GA) and particle swarm optimization (PSO) covered in this chapter are only two of the many evolutionary algorithms that have been invented.

The methods presented in this chapter do not directly address the solution of constrained problems. The assumption is that we can use penalty or filtering methods to enforce constraints. The DIRECT method is one of the few methods that handles constraints without resorting to penalties or filtering.

## Problems

7.1 Answer *true* or *false* and justify your answer.

- a) Gradient-free optimization algorithms are not as efficient as gradient-based algorithms but they converge to the global optimum.
- b) None of the gradient-free algorithms check the KKT conditions for optimality.
- c) The Nelder–Meade algorithm is a deterministic local search algorithm using heuristic criteria and direct function evaluations.
- d) The simplex is a geometric figure defined by a set of  $n$  points, where  $n$  is the dimensionality of the design variable space.
- e) The DIRECT algorithm is a deterministic global search algorithm using mathematical criteria and direct function evaluations.
- f) The DIRECT method favors small rectangles with better function values over large rectangles with worse function values.
- g) Evolutionary algorithms are stochastic global search algorithms based on heuristics and direct function evaluations.
- h) Genetic algorithms start with a population of designs that gradually decreases to a single individual design at the optimum.
- i) Each design in the initial population of a genetic algorithm should be carefully selected to ensure a successful optimization.
- j) Stochastic procedures in the genetic algorithms are necessary to maintain population diversity and therefore avoid getting stuck in local minima.
- k) Particle swarm optimization follows a model developed by biologists in the research of how bee swarms search for pollen and nectar.

- 1) All evolutionary algorithms are based on either evolutionary genetics or animal behavior.

7.2 Program the Nelder–Mead algorithm and perform the following studies:

- a) Reproduce the bean function results shown in Ex. 7.3.
- b) Add random noise to the function with a magnitude of  $10^{-4}$  using a Gaussian distribution and see if that makes a difference in the convergence of the Nelder–Mead algorithm. Compare the results to those of a gradient-based algorithm.
- c) Consider the function,

$$f(x_1, x_2, x_3) = |x_1| + 2|x_2| + x_3^2. \quad (7.29)$$

Minimize this function with the Nelder–Mead algorithm and a gradient-based algorithm. Discuss your results.

- d) *Exploration:* Study the logic of the Nelder–Mead algorithm and devise possible improvements. For example, is it a good idea to be greedier and do multiple expansions?
- 7.3 Program the DIRECT algorithm and perform the following studies:

- a) Reproduce the Jones function results shown in Ex. 7.5.
- b) Use a gradient-based algorithm with a multistart strategy to minimize the same function. On average, how many different starting points do you need to find the global minimum?
- c) Minimize the Hartmann function (defined in Appendix C.1.5) using both methods. Compare and discuss your results.
- d) *Exploration:* Develop a hybrid approach that starts with DIRECT and then switches to the gradient-based algorithm. Are you able to reduce the computational cost of DIRECT significantly while converging to the global minimum?

7.4 Program a GA algorithm and perform the following studies:

- a) Reproduce the bean function results shown in Ex. 7.10.
- b) Use your GA to minimize the Hartmann function. Estimate the rate of convergence and compare the performance of the GA with a gradient-based algorithm.

- c) Study the effect of adding checkerboard steps (Eq. 7.28) with a suitable magnitude to this function. How does this affect the performance of the GA and the gradient-based algorithm compared to the smooth case? Study the effect of reducing the magnitude of the steps.
- d) *Exploration:* Experiment with different population sizes, types of crossover, and mutation probability. Can you improve on your original algorithm? Is that improvement still observed for other problems?

7.5 Program the PSO algorithm and perform the following studies:

- a) Reproduce the bean function results shown in Ex. 7.11.
- b) Use your PSO to minimize the  $n$ -dimensional Rosenbrock function (defined in Appendix C.1.2) with  $n = 4$ . Estimate the convergence rate and discuss the performance of PSO compared to a gradient-based algorithm.
- c) Study the effect of adding noise to the objective function for both algorithms (see Prob. 7.2). Experiment with different levels of noise.
- d) *Exploration:* Experiment with different population sizes, and the values of the coefficients in Eq. 7.26. Are you able to improve the performance of your implementation for multiple problems?

7.6 Study the effect of increased problem dimensionality using the  $n$ -dimensional Rosenbrock function defined in Appendix C.1.2. Solve the problem using three approaches:

- a) Gradient-free algorithm
- b) Gradient-based algorithm with gradients computed using finite differences
- c) Gradient-based algorithm with exact gradients

You can either use an off-the-shelf optimizer or your own implementation. In each case, repeat the minimization for  $n = 2, 4, 8, 16, \dots$  up to at least 128 and see how far you can get with each approach. Plot the number of function calls required as a function of the problem dimension ( $n$ ) for all three methods on one figure. Discuss any differences in optimal solutions found by the various algorithms and dimensions. Compare and discuss your results.

Most algorithms in this book assume that the design variables are continuous. However, sometimes design variables must be discrete. Common examples of discrete optimization include scheduling, network problems, and resource allocation. This chapter introduces some techniques for dealing with discrete optimization problems.

By the end of this chapter you should be able to:

1. Identify situations where discrete variables can be avoided.
2. Convert problems with integer variables to ones with binary variables.
3. Understand the basics of various discrete optimization algorithms (branch and bound, greedy, dynamic programming, simulated annealing, binary genetic algorithms).
4. Identify which algorithms are likely to be most suitable for a given problem.

## 8.1 Binary, Integer, and Discrete Variables

Discrete optimization can be classified with three different labels: binary (sometimes called zero-one), integer, and discrete. A light switch, for example, can only be on or off and would be represented with a *binary* decision variable that is either 0 or 1. The number of wheels on a vehicle is an *integer* design variable, as it is not useful to build a vehicle with half a wheel. The material in a structure that is restricted to one of titanium, steel, or aluminum is an example of a *discrete* variable. All of these cases can be represented as integers (the discrete categories are simply mapped to integers), and an optimization problem with integer design variables is referred to as *integer programming*, *discrete optimization*, or *combinatorial optimization*.<sup>\*</sup> Problems that have both continuous and

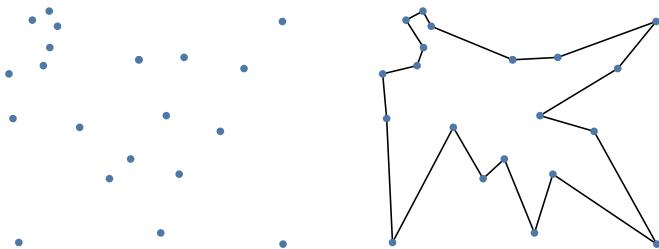
\*These phrases are often used interchangeably, but differences in meaning are intended based on the way the problem is posed.

discrete variables are referred to as *mixed integer programming* or *mixed integer optimization*.

Unfortunately, discrete optimization is NP-complete, which means that we can easily verify a solution, but there is no known approach to efficiently find a solution. Furthermore, the time required to solve the problem becomes much worse as the problem size grows.

**Example 8.1:** The drawback of an exhaustive search.

The scaling difficulty is illustrated by a well-known discrete optimization problem: the traveling salesman problem. Consider a set of cities represented graphically on the left of Fig. 8.1. The problem is to find the shortest possible route that visits each city exactly once and returns to the starting city. The right figure of Fig. 8.1 envisions one such solution (not necessarily the optimum). If there were only a handful of cities you could imagine doing an exhaustive search: enumerate all possible paths, evaluate them, and return the one with the shortest distance. Unfortunately, this is not a scalable algorithm. The number of possible paths is  $(n - 1)!$  where  $n$  is the number of cities. If, for example, we used all fifty U.S. state capitols as the set of cities, then there would be  $49! = 6.08 \times 10^{62}$  possible paths! That is an amazingly large number that could not be evaluated using an exhaustive search.



**Figure 8.1:** An example instance of the traveling salesman problem.

The apparent advantage of a discrete optimization problem is that we can construct algorithms that will find the global optimum, such as an exhaustive search. Exhaustive search ideas can also be used for continuous problems (see Section 7.4 for example, but the cost is much higher). The downside is that while an algorithm may eventually arrive at the right answer, as Ex. 8.1 highlights, in practice executing that algorithm to completion is often not practical. The goal of discrete optimization algorithms is to allow us to search the large combinatorial space more efficiently, often by using heuristics and approximate solutions.

## 8.2 Techniques to Avoid Discrete Variables

Even though a discrete optimization problem limits the options and thus conceptually sounds easier to solve, in practice discrete optimization problems are usually much more difficult and inefficient compared to continuous problems. Thus, if it is reasonable to do so, it is often desirable to find ways to avoid using discrete design variables. There are a couple ways this can be accomplished.

The first approach is an *exhaustive search*. We just discussed how exhaustive search scales poorly, but sometimes we have many continuous variables but only a few discrete variables with few options. In this case enumerating all options is possible. For each combination of discrete variables, the optimization is repeated using all continuous variables. We then choose the best feasible solution amongst all the optimization. Assuming, the continuous part of the problem can be solved, this approach will lead to the true optimum.

---

**Example 8.2:** Exhaustively evaluating discrete variables when the number of combinations is small.

Consider optimizing a propeller. While most of the design variables will be continuous, the number of blades on a propeller is not. Fortunately, the number of blades falls within a reasonably small set (2 to 6). Assuming there are no other discrete variables, we could just perform five optimizations corresponding to each option and choose the best solution.

---

A second technique is *rounding*. For some problems, we can optimize with a continuous representation, then round to integer values afterward. This is usually justifiable if the magnitude of the design variables is large or if there are many continuous variables and few discrete variables. After rounding, it is usually best to repeat the optimization once more, allowing only the continuous design variables to vary. This process may not lead to the exact optimum, and sometimes may not even lead to a feasible solution, but for many problems this is an effective approach.

One variation of this method is called *dynamic rounding*. The idea is that rather than round all continuous variables at once, perform an iterative process where you round only one, or a subset, of variables, fix them and then reoptimize using a continuous formulation. The process is repeated until all discrete variables are fixed, followed by one last optimization with the continuous variables.

Sometimes, exhaustive search is not feasible, or rounding is unacceptable as is typically the case for binary variables, or an intermediate

continuous representation is not possible. For these cases, we can utilize discrete optimization methods.

### 8.3 Branch and Bound

A popular method to solve integer optimization problems is the *branch and bound* method. It is popular not because it is the most efficient method (much better methods exist that leverage specific problem structure, some of which are discussed this chapter), but rather because it is robust and so can generally apply to a wide variety of discrete problems. This approach is particularly effective with convex integer programming problems as the method is guaranteed to find the global optimum. The most common type of convex integer problem is linear integer problems (all the objectives and constraints are linear in the design variables). The methodology can be extended to nonconvex integer optimization problems, but is generally far less effective and has no such guarantees. In this section we will assume linear mixed integer problems, with a short discussion on nonconvex problems at the end. Mathematically a linear mixed integer optimization problem can be expressed as:

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && \hat{A}x \leq \hat{b} \\ & && Ax + b = 0 \\ & && x_i \in \mathbb{Z}^+ \text{ for some or all } i \end{aligned} \tag{8.1}$$

<sup>†</sup>

#### 8.3.1 Binary Variables

Before exploring the integer case, we first explore the binary case where the discrete entries in  $x_i$  must be 0 or 1. Most integer problems can be converted to binary problems by adding additional variables and constraints. Even though the new problem is larger it is usually far easier to solve.

<sup>†</sup>Zahlen,  $\mathbb{Z}$ , is a standard symbol for the set of all integers, whereas  $\mathbb{Z}^+$  represents the set of all positive integers (including zero).

**Example 8.3:** Converting an integer problem to a binary one.

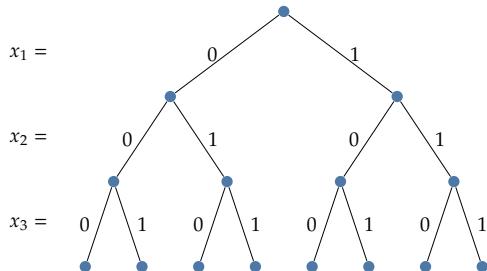
Consider a problem where an engineering device may use one of  $n$  different materials:  $y \in (1 \dots n)$ . Rather than have one design variable  $y$ , we could convert the problem to have  $n$  binary variables  $x_i$  where each  $x_i$  is 0 if material  $i$  is not selected and 1 if material  $i$  is selected. We would also need to add an additional linear constraint to make sure that one (and only one) material is

selected:

$$\sum_{i=1}^n x_i = 1 \quad (8.2)$$

The key to a successful branch and bound problem is a good *relaxation*. Relaxation means approximating an optimization problem, often by removing constraints. For a given problem many types of relaxation are possible, but for *linear* mixed integer programming problems, the most natural relaxation is to remove the integer constraints. In other words, we solve the corresponding continuous linear programming problem, also known as an LP (discussed in Section 11.2). If the solution to the original LP happened to return all binary values then we would have the solution and would terminate the search. If the LP returned fractional values then we need to branch.

Branching is done by adding additional constraints and solving additional optimization problems. For example, we could branch by adding constraints on  $x_1$ , creating two new optimization problems: the LP from above but with  $x_1 = 0$  and the LP from above but with  $x_1 = 1$ . This procedure is then repeated with additional branching as needed.



**Figure 8.2:** Enumerating the options for a binary problem with branching.

Figure 8.2 illustrates the branching concept for binary variables. If we explored all of those branches then we would be conducting an exhaustive search. The main benefit of branch and bound algorithm is that we can find ways to eliminate branches (referred to as *pruning*) to narrow down the search scope. There are two ways to prune. If any of the relaxed problems is infeasible then we know that everything from that node downward (i.e., that branch) is also infeasible. Adding more constraints cannot make an infeasible problem suddenly feasible again. Thus, that branch is pruned and we back up the tree. The other way we can eliminate branches is by determining that a better solution cannot exist on that branch. The algorithm keeps track of the best solution to the problem found so far. If one of the relaxed problems returns an objective that is worse than the best we have found

then we can prune that branch. We know this because adding more constraints will always lead to a solution that is either the same or worse, never better (assuming you always find the global optimum, which we can guarantee for LP problems). The solution from a relaxed problem provides a lower bound—the best that could be achieved if continuing on that branch. The logic for these various possibilities is summarized in Alg. 8.4. The initial starting point for  $f_{\text{best}}$  can be  $f_{\text{best}} = \infty$  if nothing is known, but if a known feasible solution exists, or can be found quickly by some heuristic, providing any finite best point can often greatly speed up the optimization.

---

**Algorithm 8.4:** Branch and bound algorithm.
 

---

**Inputs:**
 $f_{\text{best}}$ : Best known solution, if any; otherwise  $f_{\text{best}} = \infty$ 
**Outputs:**
 $x^*$ : Optimal point
 
 $f(x^*)$ : Corresponding function value
 

---

Let  $\mathcal{S}$  be the set of indices for binary constrained design variables

**while** branches remain **do**

Solve relaxed problem for  $\hat{x}, \hat{f}$

**if** relaxed problem is infeasible **then**

prune this branch, back up tree

**else**

**if**  $\hat{x}_i \in \{0, 1\} \forall i \in \mathcal{S}$  **then**

*A solution is found*

$f_{\text{best}} = \min(f_{\text{best}}, \hat{f})$ , back up tree

**else**

**if**  $\hat{f} > f_{\text{best}}$  **then**

prune this branch, back up tree

**else**

*A better solution might exist.*

branch further

**end if**

**end if**

**end if**

**end while**

---

Many variations exist for these algorithms. One design variation is the choice of which variables to branch on at a given node. One common strategy is to branch on the variable with the largest fractional component. For example, if  $\hat{x} = [1.0, 0.4, 0.9, 0.0]$  we could branch on  $x_2$  or  $x_3$  since both are fractional. We hypothesize that we are more likely to force the algorithm to make faster progress by branching on variables that are closer to midway between integers. In this case that

value would be  $x_2 = 0.4$ . Mathematically, we would choose to branch on the value closest to 0.5:

$$\min_i |x_i - 0.5| \quad (8.3)$$

Another design variation is on how to search the tree. Two common strategies are depth-first or breadth-first. A depth-first strategy continues as far down as possible (for example, by always branching left) until it cannot go further and then right branches are followed. A breadth-first strategy would explore all nodes on a given level before increasing depth. Various other strategies exist, and in general, we do not know beforehand what is best. Depth-first is a common strategy as, in the absence of other information, is likely the fastest way to find a solution (reaching the bottom of the tree generally forces a solution). Finding a solution quickly is desirable because its solution can then be used as a bound on other branches. Additionally, a depth-first strategy requires less memory storage because breadth-first must maintain a longer history as the number of levels increases, whereas depth-first only requires node storage equal to the number of levels.

---

**Example 8.5:** A binary branch and bound optimization.

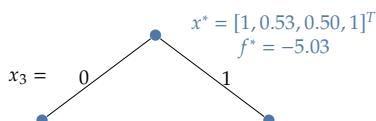
Consider the following problem:

$$\begin{aligned} & \text{minimize} && -2.5x_1 - 1.1x_2 - 0.9x_3 - 1.5x_4 \\ & \text{subject to} && 4.3x_1 + 3.8x_2 + 1.6x_3 + 2.1x_4 \leq 9.2 \\ & && 4x_1 + 2x_2 + 1.9x_3 + 3x_4 \leq 9 \\ & && x_i \in \{0, 1\} \text{ for all } i \end{aligned} \quad (8.4)$$

We begin at the first node by solving the linear relaxation. The binary constraint is removed, and instead replaced with continuous bounds:  $0 \leq x_i \leq 1$ . The solution to this LP is:

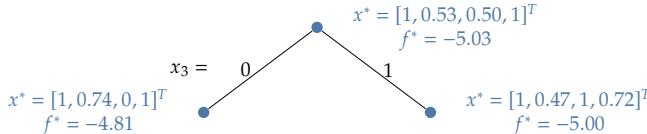
$$\begin{aligned} x^* &= [1, 0.5274, 0.4975, 1]^T \\ f^* &= -5.0279 \end{aligned} \quad (8.5)$$

There are non binary values in the solution so we need to branch. As discussed, a typical choice is to branch on the variable with the most fractional component. In this case, that is  $x_3$  so we now create two additional problems which add the constraints  $x_3 = 0$  and  $x_3 = 1$  respectively (Fig. 8.3).



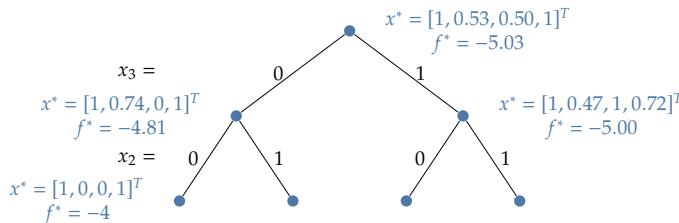
**Figure 8.3:** Initial binary branch.

While, depth-first was recommended above, for this example we will use breadth-first only because it is shorter in this case giving a more concise example. The depth-first tree is also shown at the end of the example. We solve both of the problems at this next level as shown in Fig. 8.4. Neither of these optimizations yields all binary values so we have to branch both of them. In this case the left node branches on  $x_2$  (the only fractional component) and the right node also branches on  $x_2$  (the most fractional component).



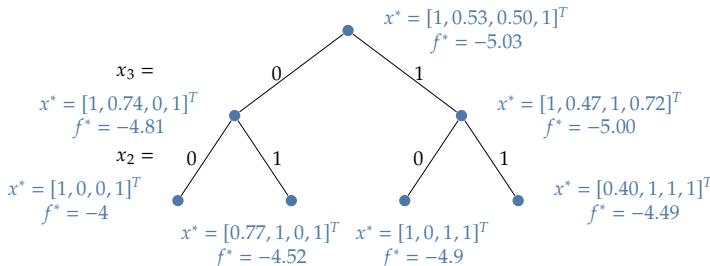
**Figure 8.4:** Solutions along these two branches.

The first branch (see Fig. 8.5) yields a feasible binary solution! The corresponding function value  $f = -4$  is saved as the best value we have seen so far. There is no need to continue on this branch as the solution cannot be improved on this particular branch.



**Figure 8.5:** The first feasible solution.

We continue solving along the rest of this row (Fig. 8.6). The third node on this row yields another binary solution. In this case the function value is  $f = -4.9$ , which is better, and so we save this as the best value we have seen so far. The second and fourth nodes do not yield a solution. Normally we'd have to branch these further but both of them have a lower bound which is worse than the best solution we have found so far. Thus, we can prune both of these branches.

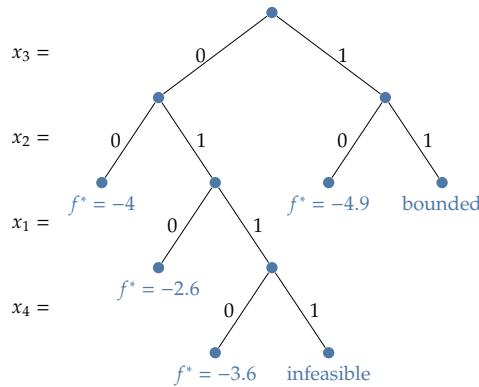


**Figure 8.6:** The rest of the solutions on this row.

All branches have been pruned and so we have solved the original problem:

$$\begin{aligned} x^* &= [1, 0, 1, 1]^T \\ f^* &= -4.9 \end{aligned} \tag{8.6}$$

Alternatively, we could have used a depth-first strategy. In this case, it is less efficient, but in general that is not known beforehand. The depth-first tree for this same example is depicted in Fig. 8.7. Feasible solutions to the problem are shown with  $f^*$ .



**Figure 8.7:** The search path with a depth-first strategy instead.

### 8.3.2 Integer Variables

If the problem cannot be put in binary form, we can use essentially the same procedure with integer variables. Instead of branching with two constraints:  $x_i = 0$  or  $x_i = 1$  we branch with two inequality constraints that will encourage solutions to find an integer solution. For example, if the variable we branched on was  $x_i = 3.4$  we would branch with two new problems with the constraints:  $x_i \leq 3$  or  $x_i \geq 4$ . An example of this is shown below.

**Example 8.6:** Branch and bound with integer variables.

Consider the following problem:

$$\begin{aligned} \text{minimize} \quad & -x_1 - 2x_2 - 3x_3 - 1.5x_4 \\ \text{subject to} \quad & x_1 + x_2 + 2x_3 + 2x_4 \leq 10 \\ & 7x_1 + 8x_2 + 5x_3 + x_4 = 31.5 \\ & x_i \in \mathbb{Z}^+ \text{ for } i = 1, 2, 3 \\ & x_4 \geq 0 \end{aligned} \tag{8.7}$$

We begin by solving the LP relaxation (i.e., we remove the integer constraints), but with a lower bound of 0. The solution to that problem is:

$$x^* = [0, 1.1818, 4.4091, 0], \quad f^* = -15.59 \quad (8.8)$$

We begin by branching on the most fractional value, which is  $x_3$ . We create two new branches:

- The original LP but with the added constraint that  $x_3 \leq 4$
- The original LP but with the added constraint that  $x_3 \geq 5$

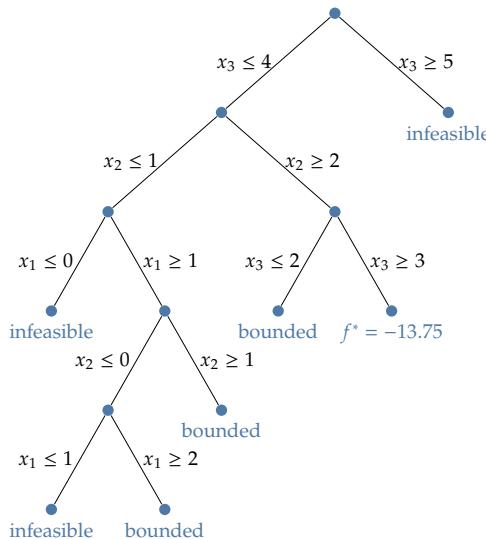
Even though depth-first is usually more efficient, we will use breadth first as it is easier to display on a figure. The solution to that first problem is:

$$x^* = [0, 1.4, 4, 0.3], \quad f^* = -15.25 \quad (8.9)$$

The second problem is infeasible so we can prune that branch.

Recall that the last variable is allowed to be continuous, so we now branch on  $x_2$  by creating two new problems with additional constraints:  $x_2 \leq 1$ , and  $x_2 \geq 2$ .

The problem continues with the same procedure, as shown in the breadth-first tree in Fig. 8.8. The figure gives some indication why solving integer problems is more time consuming than solving binary ones. Unlike the binary case, the same value is revisited with tighter constraints. For example, early on the constraint  $x_3 \leq 4$  is enforced. Later, two additional problems are created with tighter bounds on the same variable:  $x_3 \leq 2$  or  $x_3 \geq 3$ . In general, the same variable could be revisited many times as the constraints are slowly tightened, whereas in the binary case each variable is only visited once since the values can only be 0 or 1.



**Figure 8.8:** A breadth-search of the mixed integer programming example.

Once all the branches are pruned we see that the solution is:

$$\begin{aligned}x^* &= [0, 2, 3, 0.5]^T \\f^* &= -13.75.\end{aligned}\tag{8.10}$$


---

Nonconvex mixed integer problems can also be used with the branch and bound technique, and generally will use this latter strategy of forming two branches of continuous constraints. In this case the relaxed problem is not a convex problem and so we cannot provide any guarantees that we have found a lower bound for that branch. Furthermore, the cost of each suboptimization problem is increased. Thus, for nonconvex discrete problems the methodology is usually only practical for a relatively small number of discrete design variables.

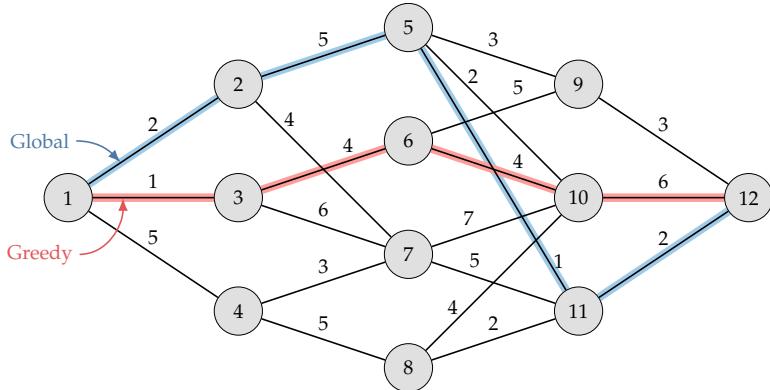
## 8.4 Greedy Algorithms

*Greedy algorithms* are perhaps the simplest approach for discrete optimization problems. This approach is more of a concept than a specific algorithm. The implementation varies with the application. The idea is to reduce the problem to a subset of smaller problems (often down to a single choice), and then make a locally optimal decision. That decision is locked in, and then the next small decision is made in the same manner. A greedy algorithm does not revisit past decisions, and so ignores much of the coupling that may occur between design variables.

**Example 8.7:** A weighted directed graph.

As an example consider the *weighted directed graph* shown in Fig. 8.9. The objective is to traverse from node 1 to node 12 with the smallest possible cost (cost denoted by the numbers above path segments). Note that a series of discrete choices must be made at each step, and those decisions limit the available options in the next step. This graph might represent a transportation problem for shipping goods, information flow through a social network, or a supply chain problem.

A greedy algorithm simply makes the best choice assuming each decision is the only decision that will be made. Starting at node 1, we first choose to move to node 3 because that is the smallest cost between the three options (node 2 cost 2, node 3 cost 1, node 4 cost 5). We then choose to move to node 6 because that is the smallest cost between the next two available options (node 6 cost 4, node 7 cost 6) and so on. The path selected by the greedy algorithm is highlighted in the figure and results in a total cost of 15. The algorithm is easy to apply and scalable, but will not generally find the global optimum. The global optimum in this case, also highlighted in the figure, results in a total



**Figure 8.9:** The greedy algorithm in this weighted directed graph results in a cost of 15, compared to the global optimum with a cost of 10.

cost of 10. To find that global optimum we have to consider the impact of our choices on future decisions. A method to do this will be discussed in the next section.

Even for a fixed problem there are, in general, many ways to construct a greedy algorithm. The advantage of this approach is that these algorithms are relatively easy to construct, and they allow us to bound the computational expense of the problem. The main disadvantages are that we usually will not find a optimal solution (in fact sometimes it can produce the worst possible solution <sup>104</sup>), and that it may not even produce a feasible solution. Despite the disadvantages there are times when the solutions, although suboptimal, are reasonably close to an optimal solution, and can be found quickly.

<sup>104.</sup> Gutin et al., *Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP*. 2002

#### Example 8.8: Some greedy algorithms.

Here are some examples of greedy algorithms for different problems.

- Traveling salesman (Ex. 8.1): Always select the nearest city as the next step.
- Propeller problem (Ex. 8.2 but with more discrete variables): optimize the number of blades with all remaining discrete variables fixed, then optimize the material selection with all remaining discrete variables fixed, . . . .
- Grocery shopping (Ex. 11.1)<sup>†</sup>: There are many possibilities for formulating a greedy solution. For example: always pick the cheapest food item next, or always pick the most nutritious food item next, or always pick the food item with the most nutrition per unit cost.

<sup>†</sup>This is a form of the knapsack problem, which is a classic problem in discrete optimization

## 8.5 Dynamic Programming

Dynamic programming is a useful technique for discrete optimization problems with a special structure. This structure also allows for usage with continuous problems and for algorithms beyond optimization. The required structure is that the problem can be posed as a *Markov chain* (for continuous problems this is called a Markov process). A Markov chain or process satisfies the Markov property, which means that a future state can be predicted from the current state without needing to know a full history. The concept can be generalized to a finite number of states (i.e., more than one but not the full history) and is called a variable-order Markov chain. If this property holds then we can break up the problem into a recursive one where a small problem is solved, and larger problems are solved by using the solutions of the smaller problems. For example, we could solve the grocery store problem for the case where the store only carried one item, then the case where the store carried two items reusing the previous solution, and so on. On the surface this may sound like a greedy optimization, but it is not. We are not using a heuristic, but fully solving the smaller problems and because of the problem structure we can reuse those solutions (we will see this in some examples shortly). This approach has become particularly useful in optimal control as well as some areas of economics and computational biology. More general design problems, like the propeller example (Ex. 8.2), do not fit this type of structure (i.e., choosing the number of blades cannot be broken up into a smaller problem separate from choosing the material).

A classic example of a Markov chain, though not an optimization problem, is the Fibonacci sequence. Its definition is:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned} \tag{8.11}$$

Notice, that we do not need a full history, but can compute the next number in the sequence just by knowing the last two states.<sup>§</sup> We could implement this with recursion as shown algorithmically in Alg. 8.9, and graphically in Fig. 8.10 for  $f_5$ .

<sup>§</sup>We can also convert this to a standard first order Markov chain by defining  $g_n = f_{n-1}$  and considering our state to be  $(f_n, g_n)$ . Then, each state only depends on the previous state.

---

Algorithm 8.9: Fibonacci with recursion.

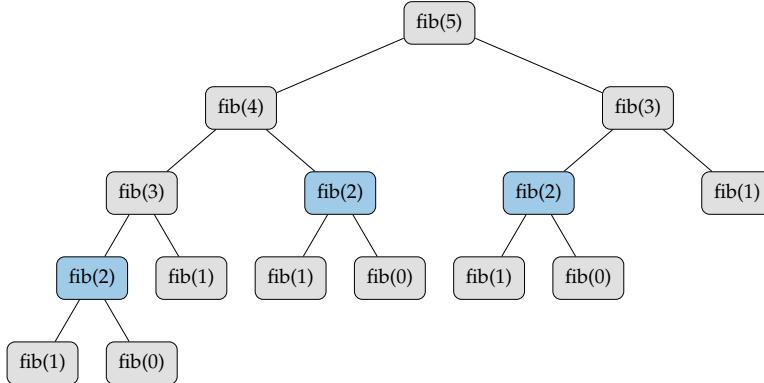
```
procedure fib(n)
  if n ≤ 1 then
    return n
```

```

else
    return fib(n - 1) + fib(n - 2)
end if
end procedure

```

---



**Figure 8.10:** Computing Fibonacci sequence using recursion. The function fib(2) is highlighted as an example to show the repetition that occurs in this recursive procedure.

While this recursive procedure works and is simple, it is inefficient. For example, the calculation for fib(2) is highlighted showing that the same calculation is repeated multiple times. There are two main approaches to avoid this inefficiency. The first is a top-down procedure called *memoization*. This just means that we store previously computed values to avoid having to compute them again. For example, the first time we need fib(2) we call the fib function and store the result (the value 1). As we progress down the tree, if we need fib(2) again we do not call the function but rather just retrieve the stored value.

More commonly we use a bottom-up approach called *tabulation*. This procedure is how one would typically show what the Fibonacci sequence looks like. We start from the bottom ( $f_0$ ) and work our way forward computing each new value using the previous states. Rather than use recursion this involves a simple loop as shown in Alg. 8.10. Whereas memoization fills entries on demand, tabulation systematically works its way up filling in entries. In either case, we reduce the computational complexity of this algorithm from exponential complexity (approximately  $O(2^n)$ ) to linear complexity ( $O(n)$ ).

---

Algorithm 8.10: Fibonacci with tabulation.

**procedure** fib2( $n$ )

$f_0 = 0$

$f_1 = 1$

```

for  $i = 2$  to  $n$  do
     $f_i = f_{i-1} + f_{i-2}$ 
end for
return  $f_n$ 
end procedure

```

---

The ideas are essentially the same in an optimization context, but before jumping into those examples we will formalize the mathematics of the approach. One main difference in optimization is that we do not have a set formula like a Fibonacci sequence. Rather at each state we will need to make a design decision, which will then change the next state. For example, with the simple graph problem shown in Fig. 8.9 we will make decisions on which path to take.

Mathematically, we express a given state as  $s_i$ , make a design decision  $x_i$ , that will transition us to the next state  $s_{i+1}$  (Fig. 8.11).

$$s_{i+1} = t_i(s_i, x_i) \quad (8.12)$$

where  $t$  is a transition function. For some variants this transition function is stochastic. At each transition we compute the cost function  $c$ . A common variation uses a discount factor on future costs. For generality we simply specify a cost function that may change at each iteration  $i$ :

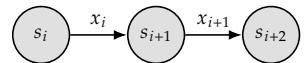
$$c_i(s_i, x_i) \quad (8.13)$$

We want to make a set of decisions that will minimize the cost not just for the current cost but for the sum of all future costs as well. This is called the *value function*  $v$ .

$$v(s_i) = \min_{x_i, \dots, x_n} [c_i(s_i, x_i) + c_{i+1}(s_{i+1}, x_{i+1}) + \dots + c_n(s_n, x_n)] \quad (8.14)$$

where  $n$  defines the time horizon we seek to optimize across. For continuous problems the time horizon may be infinite. To repeat, the value function is the *minimum* cost, not simply the cost for some arbitrary set of decisions. Note that  $v$  and  $c$  are scalar functions, but rather than use greek symbols we use  $v$  and  $c$  here as the connection to “value” and “cost” is clearer and more common.

Bellman’s *principle of optimality* notes that because of the structure of the problem (where the next state only depends on the current state), we can determine the best solution at this iteration  $x_i^*$  if we already know all the future optimal decisions  $x_{i+1}^* \dots x_n^*$ . Thus, we can recursively solve this problem from the back (bottom) determining  $x_n^*$ , then  $x_{n-1}^*$  and so



**Figure 8.11:** Diagram of state transitions in a Markov chain.

on back to  $x_i^*$ . Mathematically, this recursive procedure is captured by Bellman's equation:

$$v(s_i) = \min_{x_i} \{c(s_i, x_i) + v(s_{i+1})\}. \quad (8.15)$$

We can also express this in terms of our transition function to show the dependence on the current decision:

$$v(s_i) = \min_{x_i} \{c(s_i, x_i) + v(t_i(s_i, x_i))\}. \quad (8.16)$$

**Example 8.11:** Dynamic programming applied to graph problem.

Let us solve the graph problem posed in Ex. 8.7 using dynamic programming. For convenience, we will repeat a smaller version of the figure in Fig. 8.12. We will use the tabulation (bottom-up) approach. To do this we construct a table where we keep track of the cost to move from this node to the end (node 12), and which node we should move to next.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cost												
Next												

We start from the end. The last node is simple. There is no cost to move from node 12 to the end (we are already there) and there is no next node.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cost												0
Next												-

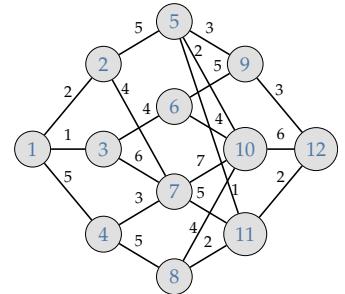
We now move back one level to consider nodes 9, 10, and 11. These nodes all lead to node 12 and so are straightforward. we will be a little more careful with the formulas as we get to the more complicated cases next.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cost									3	6	2	0
Next									12	12	12	-

We now move back one level to nodes 5, 6, 7, 8. For node 5 the cost is the following (i.e., Bellman's equation):

$$\text{cost}(5) = \min(3 + \text{cost}(9), 2 + \text{cost}(10), 1 + \text{cost}(11)) \quad (8.17)$$

Note that we have already computed the minimum value for  $\text{cost}(9)$ ,  $\text{cost}(10)$ , and  $\text{cost}(11)$  and so just look up these values in the table. In this case, the



**Figure 8.12:** Small version of Fig. 8.9 for convenience.

minimum total value is 3 and is associated with moving to node 11. Similarly, the cost for node 6 is:

$$\text{cost}(6) = \min(5 + \text{cost}(9), 4 + \text{cost}(10)) \quad (8.18)$$

The result is 8, and is realized by moving to node 9.

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cost					3	8			3	6	2	0
Next					11	9			12	12	12	-

We repeat this process, moving back and reusing optimal solutions to find the global optimum. The completed table looks like the following:

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cost	10	8	12	9	3	8	7	4	3	6	2	0
Next	2	5	6	8	11	9	11	11	12	12	12	-

From the table we see that the minimum cost is 10, and is achieved by moving to node 2, under node 2 we see that we next go to node 5, then 11, and finally 12. Thus, the tabulation gives us the global minimum for cost and the design decisions to achieve that.

---

To illustrate the concepts more generally, let us consider the knapsack problem, another classic problem in discrete optimization. In this problem we have a fixed set of items we can select from. Each item has a weight  $w_i$  and a cost  $c_i$  (since cost usually implies something to minimize we would typically use the word value here, but will stick with cost as it is consistent with our earlier notation). Our knapsack has a fixed capacity  $K$  (a scalar) then we cannot exceed. The objective is to choose the items that will yield the highest total cost subject to the capacity of our knapsack. The design variables  $x_i$  are either 1 or 0 indicating whether we take or do not take item  $i$  respectively. This problem has many variations with practical applications such as shipping, data transfer, and investment portfolio selection. Mathematically we pose the problem as:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n c_i x_i \\ & \text{subject to} && \sum_{i=1}^n w_i x_i \leq K \\ & && x_i \in \{0, 1\} \end{aligned} \quad (8.19)$$

In its present form, the problem has a linear objective and linear constraints, so branch and bound is a good fit. However, it can also be formulated as a Markov chain, so we can therefore use dynamic programming. The dynamic programming version allows us to accommodate variations such as stochasticity and other constraints more easily. To see that this can be posed as a Markov chain, we define the state as the remaining capacity of the knapsack  $k$  and the number of items we have already considered. In other words, we are interested in  $v(k, i)$  where  $v$  is the value function (optimal value given the inputs),  $k$  is the remaining capacity in the knapsack and  $i$  indicates that we have already considered items 1 through  $i$  (this doesn't mean we have added them all to our knapsack, but that we have considered them). We iterate through a series of decisions  $x_i$  deciding whether to take item  $i$  or not, which transitions us to a new state where  $i$  increases and  $k$  may decrease depending on whether or not we took the item.

The real problem we are interested in is  $v(K, n)$ , which we will solve using tabulation. Starting at the bottom, we know that  $v(k, 0) = 0$  for any  $k$ . In words, this just means that no matter what the capacity is, if we haven't considered any items yet then the value is 0. To work forward, let's consider a general case considering item  $i$ , with the assumption that we have already solved up to item  $i - 1$  for any capacity. If item  $i$  cannot fit in our knapsack ( $w_i > k$ ) then we cannot take the item. Alternatively, if the weight is less than the capacity we need to make a choice: select item  $i$  or do not. If we do not, then the value is unchanged:  $v(k, i) = v(k, i - 1)$ . If we do select item  $i$  then our value is  $c_i$  plus the best we could do with the previous items but with a capacity that was smaller by  $w_i$ :  $v(k, i) = v_i + v(k - w_i, i - 1)$ . Whichever of these decisions yields a better value is what we should choose. This process is summarized in Alg. 8.12.

---

**Algorithm 8.12:** Knapsack with tabulation.

---

**Inputs:**

$c_i$ : Cost of item  $i$   
 $w_i$ : Weight of item  $i$   
 $K$ : Total available capacity

**Outputs:**

$v(0 : K, 0 : n)$ :  $v(k, i)$  is the optimal cost for capacity  $k$  considering items 1 through  $i$ , note that indexing starts at 0

---

```

for  $k = 0$  to  $K$  do
     $v(k, 0) = 0$                                 No items considered, value is zero for any capacity
end for

```

```

for  $i = 1$  to  $n$  do Iterate forward solving for one additional item at a time
  for  $k = 0$  to  $K$  do
    if  $w_i > k$  then
       $v(k, i) = v(k, i - 1)$  Weight exceeds capacity, value unchanged
    else
       $v(k, i) = \max(v(k, i - 1), c_i + v(k - w_i, i - 1))$  Choose to either reject or take item using previous solutions
    end if
  end for
end for

```

---

Note that we will end up filling all entries in the matrix  $v[k, i]$ , in order to extract the last value  $v[K, n]$ . For small numbers, filling this matrix (or table) is often illustrated manually, hence the name tabulation. Like the Fibonacci example, using dynamic programming instead of a fully recursive solution reduces the complexity from  $O(2^n)$  to  $O(Kn)$ , which means it is pseudolinear. It is only pseudolinear because there is a dependence on the knapsack size. For small capacities the problem scales well even with many items, but as the capacity grows the problem becomes scales much less efficiently. Note that the knapsack problem requires integer weights. Real numbers can be scaled up to integers (e.g., 1.2, 2.4 becomes 12, 24). Arbitrary precision floats are not feasible given the number of combinations to search across.

**Example 8.13:** Knapsack problem with dynamic programming.

Let's consider five items with the following weights and costs:

$$\begin{aligned} w_i &= [4, 5, 2, 6, 1] \\ c_i &= [4, 3, 3, 7, 2] \end{aligned} \tag{8.20}$$

The capacity of our knapsack is  $K = 10$ . Using Alg. 8.12 we find that the optimal cost is 12. The value matrix looks as follows:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 3 & 3 & 3 \\ 0 & 0 & 0 & 3 & 3 & 5 \\ 0 & 4 & 4 & 4 & 4 & 5 \\ 0 & 4 & 4 & 4 & 4 & 6 \\ 0 & 4 & 4 & 7 & 7 & 7 \\ 0 & 4 & 4 & 7 & 7 & 9 \\ 0 & 4 & 4 & 7 & 10 & 10 \\ 0 & 4 & 7 & 7 & 10 & 12 \\ 0 & 4 & 7 & 7 & 11 & 12 \end{bmatrix} \tag{8.21}$$

To determine which items produce this cost we need to add a bit more logic. To focus on the main principles this was left out of the previous algorithm, but for completeness is discussed in this example. To keep track of the selected items we need to define a selection matrix  $S$  of the same size as  $v$  (note that this matrix is indexed starting at zero in both dimensions). Every time we accept an item  $i$  in Alg. 8.12 we note that in the matrix as  $S_{k,i} = 1$ . We would replace this line:

$$v(k, i) = \max(v(k, i - 1), c_i + v(k - w_i, i - 1))$$

with

**if**  $c_i + v(k - w_i, i - 1) > v(k, i - 1)$  **then**

$$v(k, i) = c_i + v(k - w_i, i - 1)$$

$$S(k, i) = 1$$

**else**

$$v(k, i) = v(k, i - 1)$$

**end if**

Then at the end of the algorithm we can determine, which entries were saved by this logic:

**Input:**  $S$  *Selection matrix ( $K + 1 \times n + 1$ ) matrix from (8.12)*  
 $k = K$   
 $X^* = \{\}$  *Initialize solution  $X^*$  as an empty set*  
**for**  $i = n$  **to** 1 **by**  $-1$  **do**  
  **if**  $S_{k,i} = 1$  **then**  
    add  $i$  to  $X^*$  *Item  $i$  was selected*  
     $k -= w_i$   
  **end if**  
**end for**  
**return**  $X^*$

For this example the selection matrix  $S$  looks as follows:

$$S = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \quad (8.22)$$

Following the above algorithm we find that we selected items 3, 4, 5 for a total cost of 12, as expected, and a total weight of 9.

Like greedy algorithms, dynamic program is more of a technique

than a specific algorithm. The implementation will in general vary with the particular application.

## 8.6 Simulated Annealing

Simulated annealing<sup>¶</sup> is a methodology designed for discrete optimization problems, although it can also be effective for continuous multimodal problems as we will discuss. Inspiration for the algorithm comes from the annealing process of metals. The atoms in a metal form a crystal lattice structure. If the metal is heated the atoms move around freely. As the metal is cooled down, the atoms slow down and if the cooling is slow enough they reconfigure into a minimum energy state. Alternatively, if the metal is quenched, or cooled rapidly, the metal recrystallizes with a different higher energy state (called an amorphous metal).

From statistical mechanics, the Boltzman distribution (also called Gibbs distribution) describes the probability of a system occupying a given energy state:

$$\text{prob}(e) \propto \exp\left(\frac{-e}{k_B T}\right) \quad (8.23)$$

where  $e$  is the energy level,  $T$  is the temperature, and  $k_B$  is Boltzmann's constant. This equation shows that as the temperature is decreased, the probability of occupying a higher energy state is decreased, but it is not zero. Therefore, unlike classical mechanics, an atom could jump to a higher energy state, with some small probability. It is this property, applied to an optimization algorithm, that gives the methodology an exploratory nature avoiding premature convergence to a local minimum. The temperature level provides some control on the level of expected exploration.

In the Metropolis algorithm the probability of transitioning from energy state  $e_1$  to energy state  $e_2$  is taken to be:

$$\text{prob} = \exp\left(\frac{-(e_2 - e_1)}{k_B T}\right) \quad (8.24)$$

If  $e_2 < e_1$  then the predicted probability would be greater than 1, and so is capped at 1.

In the optimization analogy, the energy level is our objective function. Temperature is a parameter controlled by the optimizer, which begins high and is slowly "cooled" to drive towards convergence. At a given iteration the design variables are given by  $x$ , and the objective (or energy) is given by  $f(x^{(i)})$ . A new state  $x_{\text{new}}$  is selected at random in the neighborhood of  $x$ . If the energy level decreases then the new state

<sup>¶</sup>First developed by Kirkpatrick *et al.*<sup>105</sup> and Černý<sup>106</sup>.

<sup>105</sup> Kirkpatrick *et al.*, *Optimization by Simulated Annealing*. 1983

<sup>106</sup> Černý, *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm*. 1985

is accepted. If the energy level increases, the new state might still be accepted with probability,

$$\exp\left(\frac{-(f(x_{\text{new}}) - f(x))}{T}\right), \quad (8.25)$$

where Boltzmann's constant is removed because it is just an arbitrary scale factor in the optimization context. Otherwise, the state remains unchanged. Constraints can be naturally handled in this algorithm without resorting to penalties by rejecting any infeasible step.

We must supply the optimizer with a function that provides a random *neighboring* design from the set of possible design configurations. A neighboring design is usually related to the current design, as opposed to picking a pure random design from the entire set. In defining the neighborhood structure, one might wish to define transition probabilities so that all neighbors are not equally likely. This type of structure is common in Markov chain problems.

Finally, we need to determine the *annealing schedule* (or *cooling schedule*), a process for decreasing the temperature throughout the optimization. A common approach is exponential decrease:

$$T = T_0 \alpha^k \quad (8.26)$$

where  $T_0$  is the initial temperature,  $\alpha$  the cooling rate, and  $k$  is the iteration number. The cooling rate  $\alpha$  is a number close to one, like 0.8 – 0.99. Another simple approach to iterate towards zero temperature is:

$$T = T_0 \left(1 - \frac{k}{k_{\max}}\right)^{\beta} \quad (8.27)$$

where the exponent  $\beta$  is usually in the range of 1–4, with higher exponents spending more time at low temperatures. In many approaches the temperature is kept constant for a fixed number of iterations (or a fixed number of successful moves) before moving to the next decrease. Many methods are simple schedules with a predetermined rate, although more complex adaptive methods also exist.<sup>107</sup> The annealing schedule can have a strong impact on the algorithm's performance and some experimentation is required to select an appropriate schedule for the problem at hand. The important principles are that the temperature should start high enough to allow for exploration, significantly higher than the maximum expected energy change (change in objective) but not so high that computational time is wasted with lots of random searching, and that cooling should occur slowly to improve the ability to recover from local optimum, imitating the annealing process as opposed to the quenching process.

<sup>107</sup> See for example Andresen *et al.*<sup>107</sup>

107. Andresen *et al.*, *Constant thermodynamic speed for minimizing entropy production in thermodynamic processes and simulated annealing*. 1994

The basic algorithm is summarized in Alg. 8.14 where for simplicity in the description the annealing schedule uses an exponential decrease at every iteration.

---

**Algorithm 8.14:** Simulated Annealing
 

---

**Inputs:** $x^{(0)}$ : *Starting point* $T^{(0)}$ : *Initial temperature***Outputs:** $x^*$ : *Optimal point*


---

```

for  $k = 0$  to  $k_{\max}$  do           Simple iteration; convergence metrics can be used instead.
     $x_{\text{new}} = \text{neighbor}(x^{(k)})$           Randomly generate from neighbors
    if  $f(x_{\text{new}}) \leq f(x^{(k)})$  then      Energy decreased, jump to new state
         $x^{(k+1)} = x_{\text{new}}$ 
    else
         $r \in \mathbb{U}[0, 1]$                       Randomly draw from uniform distribution
         $p = \exp\left(\frac{-(f(x_{\text{new}}) - f(x^{(k)}))}{T}\right)$ 
        if  $p \geq r$  then                    Probability high enough to jump
             $x^{(k+1)} = x_{\text{new}}$ 
        else
             $x^{(k+1)} = x^{(k)}$                   Otherwise remain at current state
        end if
    end if
     $T = \alpha T$                           Reduce temperature
end for

```

---



---

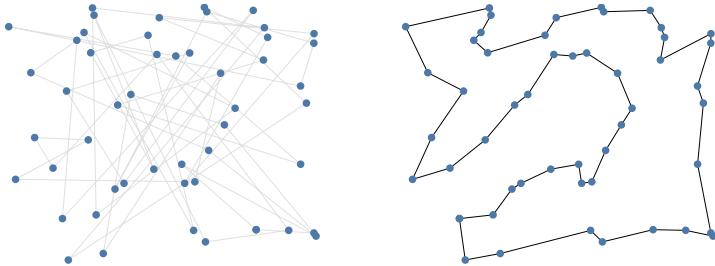
**Example 8.15:** Traveling Salesman with Simulated Annealing
 

---

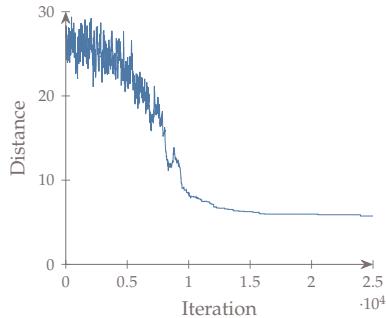
This example sets up the traveling salesman problem with 50 points randomly distributed (from uniform sampling) on a square grid with sides of length 1 (top of Fig. 8.13). The objective is the total Euclidian distance of a path that traverses all points and returns to the starting point. The design variables are simply a sequence of integers that correspond to the order in which to traverse the points. We generate new neighboring designs using the technique from Lin<sup>108</sup>, where one of two options is randomly chosen at each iteration: 1) randomly choose two points and flip the direction of the path segments between those two points, or 2) randomly choose two points and move the path segments to follow another randomly chosen point. The distance traveled by the randomly generated initial set of points is 26.2. We specify an iteration budget of 25,000 iterations, set the initial temperature to be 10, and every 100 iterations we decrease the temperature by a multiplicative

<sup>108</sup>. Lin, *Computer Solutions of the Traveling Salesman Problem*. 1965

factor of 0.95. The final design is shown in the bottom of Fig. 8.13 with a path length of 5.61. The final path might not be *the* global optimum (remember these finite time methods are only approximations of the full combinatorial search), but the methodology is effective and fast for this problem in finding at least a near-optimal solution. The iteration history is shown in Fig. 8.14.



**Figure 8.13:** Initial and final path for traveling salesman problem.



**Figure 8.14:** Convergence history of the simulated annealing algorithm.

---

The simulated annealing algorithm can be applied to continuous multimodal problems as well. The motivation is similar in that the initial high temperature would permit the optimizer to escape local minima whereas a pure descent-based approach would not. By slowly cooling, the initial exploration gives way to exploitation. The only real change in the procedure is in the neighbor function. A typical approach is to generate a random direction, and choose a step size proportional to the temperature. Thus, smaller, more conservative steps are taken as the algorithm progresses. If bound constraints are used, they would be enforced at this step. Purely random step directions are not particularly efficient for many continuous problems, particularly when most directions are bad (e.g., a narrow valley, or near convergence). One variation adopts concepts from the Nelder Mead simplex (Section 7.3) to improve efficiency.<sup>109</sup> Overall, simulated annealing has made more impact on discrete problems as compared to

<sup>109</sup>. Press et al., *Numerical Recipes in C: The Art of Scientific Computing*. 1992

continuous ones.

## 8.7 Quantum Annealing

Quantum annealing is similar to simulated annealing but borrows ideas from quantum mechanics instead of from statistical mechanics. While simulated annealing allows for a design to probabilistically jump over an energy barrier based on the concept of thermal energy, quantum annealing allows for a design to probabilistically tunnel through an energy barrier based on the Heisenberg uncertainty principle. In brief, one major difference is that in simulated annealing the probability of accepting a worse step is related to the change in function value, whereas in quantum annealing the probability of accepting a worse step is related to the change in function value and the change in design variables (Fig. 8.15). Thus, the intuition is that a function space with local minimum that are deep (high thermal barrier), but close to other minima (small tunneling distance), may be more suitable for quantum annealing as opposed to simulated annealing.

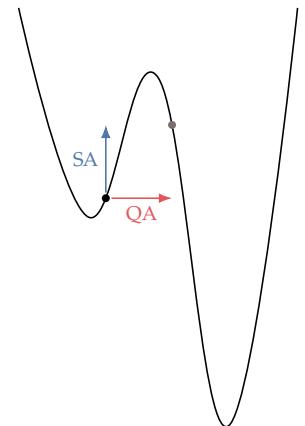
While the algorithm can be represented classically, it is more efficiently implemented on a quantum computer. A set of entangled quantum bits (qubits) already possess the desired properties for the algorithm, allowing these problems to be solved *very* quickly.

## 8.8 Binary Genetic Algorithms

The binary form of a genetic algorithm (GA) can be directly used with discrete variables. Since the binary form requires a discrete representation for the members of the population anyway, using discrete design variables is a natural fit. The details of this method were discussed in Section 7.5.1.

## 8.9 Summary

This chapter discussed various strategies to approach discrete optimization problems. Some problems can be well approximated using rounding, or only have a few discrete combinations allowing for explicit enumeration. For problems that can be posed as linear, branch and bound is very effective. If the problem can be posed as a Markov chain then dynamic programming is a useful technique. If these categorizations are not applicable then a stochastic method like simulated annealing or genetic algorithms may work well. These stochastic methods typically struggle as the dimensionality of the problem increases



**Figure 8.15:** Illustration of main difference between simulated annealing (SA) and quantum annealing (QA). The black dot is the starting point, and the light dot a candidate point. The probability that SA takes the worse step is related to the function increase, whereas the probability that QA accepts the worse step is related to both the function increase and the magnitude of the design variable changes.

(although some simulated annealing problems can scale better if there are clever ways to quickly evaluate designs in the neighborhood, as is done with the traveling salesman problem). An alternative to these various algorithms is to use a greedy strategy, which can scale well, but is a heuristic usually requiring some loss in solution quality.

## Problems

8.1 Answer *true* or *false* and justify your answer.

- a) All discrete variables can be represented by integers.
- b) Discrete optimization algorithms sometimes use heuristics and find only approximate solutions.
- c) The rounding technique solves a discrete optimization problem with continuous variables and then rounds each resulting design variable, objective, and constraint to the nearest integer.
- d) Exhaustive search is the only way to be sure we have found the global minimum for a problem that involves discrete variables.
- e) The branch and bound method is guaranteed to find the global optimum for convex problems.
- f) When using the branch and bound method for binary variables, the same variable might have to be revisited.
- g) When using the branch and bound method, the breath-first strategy requires less memory storage than the depth-first strategy.
- h) Greedy algorithms never reconsider a decision once it has been made.
- i) The Markov property applies when a future state can be predicted from the current state without needing to know any previous state.
- j) Both memoization and tabulation reduce the computational complexity of dynamic programming such that it no longer scales exponentially.
- k) Simulated annealing can be used to minimize smooth unimodal functions of continuous design variables.
- l) Simulated annealing, genetic algorithms, and dynamic programming include stochastic procedures.

8.2 *Converting to binary variables.* You have one integer design variable  $x \in [1, n]$ . Let's say, for example, that this variable represents one of  $n$  materials that we would like to select from. Convert this to an equivalent binary problem so that it is more efficient for branch and bound. To accomplish this you will need to create additional design variables and additional constraints.

8.3 *Branch and bound.* Solve the following problem using a *manual* branch and bound approach (i.e., show each LP subproblem) as is done in Ex. 8.5.

$$\begin{aligned} & \text{maximize} && 0.5x_1 + 2x_2 + 3.5x_3 + 4.5x_4 \\ & \text{subject to} && 5.5x_1 + 0.5x_2 + 3.5x_3 + 2.3x_4 \leq 9.2 \\ & && 2x_1 + 4x_2 + 2x_4 \leq 8 \\ & && 1x_1 + 3x_2 + 3x_3 + 4x_4 \leq 4 \\ & && x_i \in \{0, 1\} \text{ for all } i \end{aligned} \tag{8.28}$$

8.4 *Solve an integer linear programming problem.* A chemical company produces four types of products: A, B, C, and D. Each requires labor to produce and uses some combination of chlorine, sulfuric acid, and sodium hydroxide in the process. The production process can also produce these chemicals as a byproduct, rather than just consuming them. The chemical mixture and labor required for the production of the three products are listed in the table below, along with the availability per day. The market value for one barrel of A, B, C, and D are \$50, \$30, \$80, and \$30 respectively. Determine the number of barrels of each to produce in order to maximize profit using three different approaches:

- a) As a continuous linear programming problem with rounding.
- b) As an integer linear programming problem.
- c) Exhaustive search.

	A	B	C	D	Limit
Chlorine	0.74	-0.05	1.0	-0.15	97
Sodium hydroxide	0.39	0.4	0.91	0.44	99
Sulfuric acid	0.86	0.89	0.09	0.83	52
Labor (person-hours)	5	7	7	6	1000

8.5 *Solve a dynamic programming problem.* Solve the knapsack problem with the following weights and costs:

$$\begin{aligned} w_i &= [2, 5, 3, 4, 6, 1] \\ c_i &= [5, 3, 1, 5, 7, 2] \end{aligned} \tag{8.29}$$

and a capacity of  $K = 12$ . Maximize the cost subject to capacity constraint. Use the following two approaches:

- a) a greedy algorithm where you take the item with the best cost/weight ratio (that fits within the remaining capacity) at each iteration.
- b) dynamic programming

8.6 *Simulated annealing.* Construct a traveling salesman problem with 50 randomly generated points. Implement a simulated annealing algorithm to solve.

8.7 *Binary genetic algorithm.* Solve the same problem as above (traveling salesman) with a binary genetic algorithm.

8.8 *Binary genetic algorithm II.* Something more suited to a GA.

Up to this point in the book, all of our optimization problem formulations have had a single objective function. In this chapter, we consider *multiobjective* optimization problems, that is, problems whose formulations have more than one objective function. A classic example of multiobjective optimization is risk versus reward in financial investments.

By the end of this chapter you should be able to:

1. Determine the scenarios where multiobjective optimization is useful.
2. Understand the concept of dominance and identify a Pareto set.
3. Use and identify various methods for performing multi-objective optimization and understand the pros and cons of the methods.

## 9.1 Multiple Objectives

Before discussing how to solve multiobjective problems we must first explore what it means to have more than one objective. In some sense, there is no such thing as a multiobjective optimization problem. While many metrics may be important to the design engineer, in practice only one thing can be made best at a time. A common technique when presented with multiple objectives, as we will discuss in more detail, is to assign weights to the various objectives and combine them. But in doing so we have defined a single objective. More generally, multiobjective optimization is useful in exploring tradeoffs between different metrics, but if we intend to select a design (or even multiple designs) from the presented option, we have indirectly formulated an objective. This new objective may be difficult to formalize, and we

should be careful that the imprecision in our selection is warranted.

---

**Tip 9.1:** Are you sure you have multiple objectives?

A common pitfall for beginner optimization practitioners is to categorize a problem as multiobjective without critical evaluation. When considering whether you should use more than one objective, you should ask whether or not there is a more fundamental underlying objective, or if some of the “objectives” are actually constraints. Solving a multiobjective problem is much more costly than solving a single objective one, so you should make sure you absolutely need multiple objectives.

---

---

**Example 9.2:** Selecting an objective.

Determining the appropriate objective is often a real challenge. For example, in designing an aircraft, one may decide that minimizing drag and minimizing weight are both important. However, these metrics are competing and cannot be minimized simultaneously. Instead, we may conclude that maximizing range (the distance the aircraft can fly) is the underlying metric that matters most for our application and appropriately balances the tradeoffs between weight and drag. Or, perhaps maximizing range isn’t the right metric. Range may be important, but only insofar as we reach some threshold. Increasing the range does not increase the value because range is a constraint. The underlying objective in this scenario may be some other metric like operating costs.

---

Despite these considerations, there are still good reasons to pursue a multiobjective problem. A few of the most common reasons include:

1. Multiobjective optimization allows us to quantify tradeoff sensitivities between different objectives and constraints. The benefits of this approach will become apparent when we discuss Pareto surfaces and can lead to important design insights.
2. Multiobjective optimization provides a “family” of designs rather than a single design. A family of options is desirable when decision making needs to be deferred to a later stage as more information is gathered. For example, an executive team or higher-fidelity numerical simulations may be used to make later design decisions.
3. For some problems, the underlying objective is either unknown or too difficult to compute. For example, cost and environmental impact may be two important metrics for a new design. While

the latter could arguably be turned into a cost, doing so may be too difficult to quantify and add an unacceptable amount of uncertainty.

Mathematically, the only change to our optimization problem formulation is that the objective statement,

$$\text{minimize } f(x) \quad (9.1)$$

becomes

$$\text{minimize } f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_{n_f}(x) \end{bmatrix}, \text{ where } n_f \geq 2 \quad (9.2)$$

The constraints are unchanged, unless some of them have been reformulated as objectives. This multiobjective formulation might require tradeoffs when trying to minimize all functions simultaneously because at some point, further reduction in one objective can only be achieved by increasing one or more of the other objectives.

One exception occurs if the objectives are independent because they depend on different sets of design variables. Then, the objectives are said to be *separable* and they can be minimized independently. If there are constraints, these need to be separable as well. However, separable objectives and constraints are rare because in real engineering systems all functions tend to be linked in some way.

Given that multiobjective optimization requires tradeoffs, we need a new definition of optimality. In the next section, we explain how there are an infinite number of points that are optimal, forming a surface in the space of objective functions. After defining optimality for multiple objectives, we present several possible methods for solving multiobjective optimization problems.

## 9.2 Pareto Optimality

With multiple objectives, we have to reconsider what it means for a point to be optimal. In multiobjective optimization we use the concept of *Pareto optimality*.

Figure 9.1 shows three designs measured against two objectives that we want to minimize:  $f_1$  and  $f_2$ . Let us first compare design A with design B. From the figure, we see that design A is better than design B in both objectives. In the language of multiobjective optimization we say that design A *dominates* design B. One design is said to dominate another

design if it is superior in all of the objectives (design A dominates any design in the shaded rectangle). Comparing design A with design C, we note that design A is better in one objective ( $f_1$ ), but worse in the other objective ( $f_2$ ). Neither design dominates the other.

A point is said to be *nondominated* if none of the other evaluated points dominate it. If a point is nondominated by any point in the entire domain, then that point is called *Pareto optimal*. This does not imply that this point dominates all other points; it simply means no other point dominates it. The set of all Pareto optimal points is called the *Pareto set*, and is visualized in Fig. 9.2. The Pareto set refers to the vector of points  $x^*$ , whereas the Pareto front refers to the vector of functions  $f(x^*)$ .

---

**Example 9.3:** A Pareto front in wind farm optimization.

---

The Pareto front is a useful tool to produce design insights. Figure 9.3 shows a notional Pareto front for a wind farm optimization. The two objectives are maximizing power production (shown with a negative sign so that it is minimized), and minimizing noise. The Pareto front is helpful to understand tradeoff sensitivities. For example, the left end point shows the maximum power solution, and the right end point shows the minimum noise solution. The nature of the curve on the left side tells us how much power we have to sacrifice for a given reduction in noise. If the slope is steep, as is the case in the figure, we can see that a small sacrifice in maximum power production can be exchanged for greatly reduced noise. However, if even larger noise reductions are sought then large power reductions will be required. Conversely, if the left side of the figure had a flatter slope we would know that small reductions in noise would require significant decreases in power. Understanding the magnitude of these tradeoff sensitivities is helpful in making high-level design decisions.

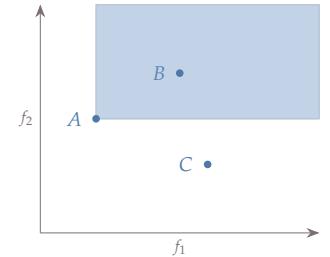
---

## 9.3 Solution Methods

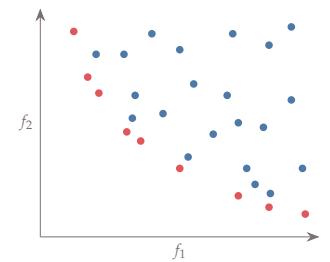
Various solution methods exist to solving multiobjective problems. This chapter does not cover all methods, but highlights some of the more commonly used methods. These included the weighted-sum method, the  $\epsilon$ -constraint method, the normal boundary interface method, and evolutionary algorithms.

### 9.3.1 Weighted Sum

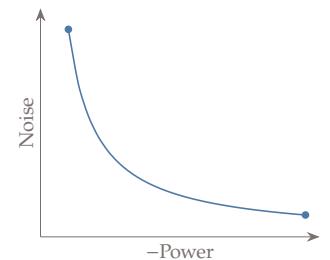
The weighted-sum method is easy to use, but it is not particularly efficient. Other methods exist that are just as simple but have better performance. It is only introduced because it is well known and is



**Figure 9.1:** The three designs: A, B, and C, are plotted against two objectives:  $f_1$  and  $f_2$ . The region in the shaded rectangle highlights points that are dominated by design A.



**Figure 9.2:** A plot of all the evaluated points in the design space plotted against two objectives  $f_1$  and  $f_2$ . The set of red points are not dominated by any other and thus form the Pareto set.



**Figure 9.3:** A notional Pareto front representing power and noise trade-offs for a wind farm optimization problem.

frequently used. The idea is to combine all of the objectives into one objective using a weighted sum, which can be written as:

$$\bar{f}(x) = \sum_i^N w_i f_i(x), \quad (9.3)$$

where  $N$  is the number of objectives and the weights are usually normalized such that

$$\sum_i^N w_i = 1 \quad (9.4)$$

If we have two objectives, the objective reduces to

$$\bar{f}(x) = wf_1(x) + (1 - w)f_2(x), \quad (9.5)$$

where  $w$  is a weight in  $[0, 1]$ .

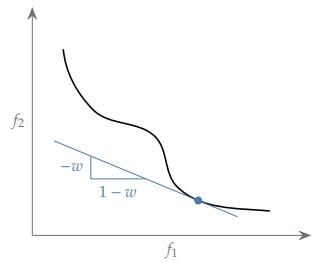
Consider a two-objective case. Points on the Pareto set are determined by choosing a weight  $w$ , completing the optimization for the composite objective, and then repeating the process for a new value of  $w$ . It is straightforward to see that at the extremes  $w = 0$  and  $w = 1$ , the optimization returns the designs that optimize one objective while ignoring the other. The weighted-sum objective forms an equation for a line with the objectives as the ordinates. Conceptually we can think of this method as choosing a slope for the line (by selecting  $w$ ), then pushing that line down and to the left as far as possible until it is just tangent to the Pareto front (Fig. 9.4). With the above form of the objective the slope of the line would be:

$$\frac{df_2}{df_1} = \frac{-w}{1-w} \quad (9.6)$$

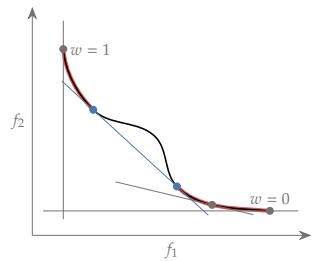
This procedure identifies one point in the Pareto set and the procedure must then be repeated with a new slope.

The main benefit of this method is that it is easy to use. However, the drawbacks are that 1) uniform spacing in  $w$  leads to nonuniform spacing along the Pareto set, 2) it is not obvious which values of  $w$  should be used to sweep out the Pareto set effectively, and 3) this method can only return points on the convex portion of the Pareto front (see Fig. 9.5).

Using the Pareto front shown in Fig. 9.4, Fig. 9.5 highlights the convex portion of the Pareto front. Those are the only portions of the Pareto front that can be found using a weighted sum method.



**Figure 9.4:** The weighted-sum method defines a line for each value of  $w$  and find the point tangent to the Pareto front.



**Figure 9.5:** The convex portion of this Pareto front are the portions highlighted.

### 9.3.2 Epsilon-constraint method

The  $\epsilon$ -constraint method works by minimizing one objective, while setting all other objectives as additional constraints<sup>110</sup>

$$\begin{aligned} & \text{minimize } f_i \\ & \text{by varying } x \\ & \text{subject to } f_j \leq \epsilon_j \quad \text{for all } j \neq i, \\ & \quad g(x) \leq 0 \\ & \quad h(x) = 0 \end{aligned} \tag{9.7}$$

Then, we must repeat this procedure for different values of  $\epsilon_j$ .

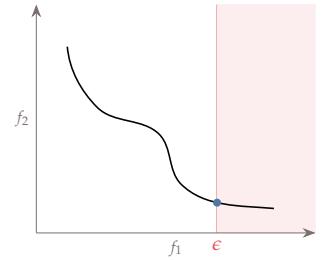
This method is visualized in Fig. 9.6. In this example, we constrain  $f_1$  to be less than a certain value and minimize  $f_2$  to find the corresponding point on the Pareto front. We then repeat this procedure for different values of  $\epsilon$ .

One advantage of this method is that determining appropriate values for  $\epsilon$  is more intuitive than selecting the weights in the previous method, although one must be careful to choose values that result in a feasible problem. Another advantage is that this method reveals the non-convex portions of the Pareto front. Its main limitation is that like the weighted-sum method, a uniform spacing in  $\epsilon$  does not in general yield uniform spacing of the Pareto front and therefore it might still be inefficient, particularly with more than two objectives.

### 9.3.3 Normal Boundary Intersection

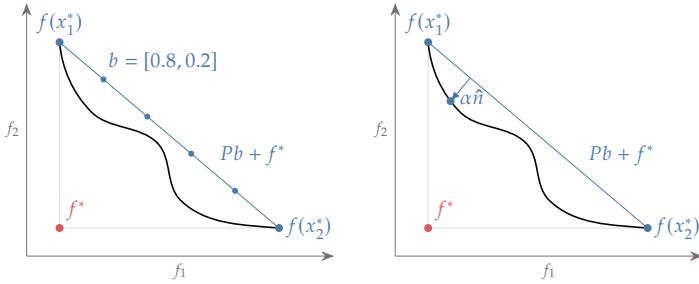
The normal boundary intersection method is designed to address the issue of nonuniform spacing along the Pareto front.<sup>111</sup> The basic idea is to first find the extremes of the Pareto set; in other words, we minimize the objectives one at a time. These extreme points are referred to as *anchor points*. Next, construct a plane that passes through the anchor points. We space points along this plane (usually evenly) and starting from those points, solve optimization problems that search along lines normal to this plane.

This procedure is shown in Fig. 9.7 for a two objective case. In this case, the plane that passes through the anchor points is a line. We now space points along this plane by choosing a vector of weights that we will call  $b$ , and which are illustrated on the left hand figure. The weights are constrained such that  $b_i \in [0, 1]$ , and  $\sum_i b_i = 1$ . If we make  $b_i = 1$  and all other entries zero, then this equation returns one of the anchor points,  $f(x_i^*)$ . For two objectives, we would define  $b$  as  $b = [w, 1 - w]^T$  and vary  $w$  in equal steps between 0 and 1.



**Figure 9.6:** The vertical line represents an upper bound constraint on  $f_1$ . The other objective  $f_2$  is minimized to reveal one point in the Pareto set. This procedure is then repeated for different constraints on  $f_1$  to sweep out the Pareto set.

<sup>111</sup> Das et al., *Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems*. 1998



**Figure 9.7:** A notional example of the normal boundary intersection method. A plane is created passing through the single objective optima, and solutions are sought normal to that plane to allow for a more evenly spaced Pareto front.

Starting with a specific value of  $b$ , we search along a line perpendicular to this point, represented by the line with the arrow seen on the right hand figure. We seek to find the point along this line that is furthest away from the plane (a maximization problem), with the constraint that the point is consistent with the objective functions. The resulting optimal point found along this line is shown as a point along the Pareto front. We then repeat for another set of weighting parameters in  $b$ .

We can see how this method is similar to the constraint epsilon method, but instead of searching along lines parallel to one of the axes, we search along lines parallel to this plane. The idea is that even spacing along this plane is more likely to lead to even spacing along the Pareto front.

Mathematically, we start by determining the anchor points, which are just single objective optimization problems. From the anchor points we define what is called the *utopia* point. The utopia point is an ideal point that cannot be obtained, where every objective reaches its minimum simultaneously (shown in the lower left corner of Fig. 9.7):

$$f^* = \begin{bmatrix} f_1(x_1^*) \\ f_2(x_2^*) \\ \vdots \\ f_n(x_n^*) \end{bmatrix}, \quad (9.8)$$

where  $x_i^*$  denotes the design variables that minimize objective  $f_i$ . The utopia point allows us to define the equation of a plane that passes through all anchor points,

$$Pb + f^*, \quad (9.9)$$

where the  $i^{\text{th}}$  column of  $P$  is  $f(x_i^*) - f^*$ . A single vector  $b$ , whose length is given by the number of objectives, defines a point on the plane.

We now define a vector  $(\hat{n})$  that is normal to this plane, in the direction toward the origin. We search along this vector using a step length  $\alpha$ , yielding

$$Pb + f^* + \alpha \hat{n}. \quad (9.10)$$

Computing the exact normal ( $\hat{n}$ ) is involved and it is not actually necessary that the vector be exactly normal. As long as the vector points toward the Pareto front then it will still yield well-spaced points. In practice, a quasi-normal vector is often used, such as,

$$\tilde{n} = -P\mathbf{1} \quad (9.11)$$

where  $\mathbf{1}$  is a vector of ones.

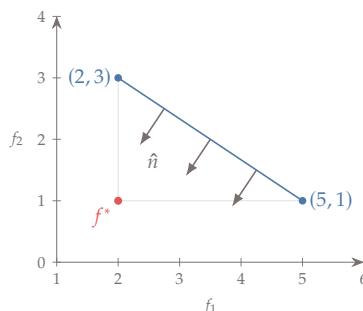
We now solve the following optimization problem, for a given vector  $b$ , to yield a point on the Pareto front:

$$\begin{aligned} & \text{maximize} && \alpha \\ & \text{by varying} && x, \alpha \\ & \text{subject to} && Pb + f^* + \alpha\hat{n} = f(x) \\ & && g(x) \leq 0 \\ & && h(x) = 0 \end{aligned} \quad (9.12)$$

This means that we are finding the point furthest away from the anchor point plane, starting from a given value for  $b$ , while satisfying the original problem constraints. The process is then repeated for additional values of  $b$  to sweep out the Pareto front.

In contrast to the previously mentioned methods, this method yields a more uniformly spaced Pareto front, which is desirable for computational efficiency, albeit at the cost of a more complex methodology.

**Example 9.4:** A two-dimensional normal boundary interface problem.



**Figure 9.8:** Search directions are normal to the line connecting anchor points.

First, we optimize the objectives one at a time, which in our example results in the two anchor points shown in Fig. 9.8:  $f(x_1^*) = (2, 3)$  and  $f(x_2^*) = (5, 1)$ . The utopia point is then:

$$f^* = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad (9.13)$$

For the matrix  $P$  recall that the  $i^{\text{th}}$  column of  $P$  is  $f(x_i^*) - f^*$ :

$$P = \begin{bmatrix} 0 & 3 \\ 2 & 0 \end{bmatrix} \quad (9.14)$$

Our quasi-normal vector is given by  $-P\mathbf{1}$  (note that the true normal is  $[-2, -3]$ ):

$$\tilde{n} = \begin{bmatrix} -3 \\ -2 \end{bmatrix} \quad (9.15)$$

We now have all the parameters we need to solve Eq. 9.12.

For most multiobjective design problems additional complexity beyond the normal boundary intersection (NBI) method is unnecessary, however, even this method can still have deficiencies for problems with unusual Pareto fronts and new methods continue to be developed. For example, the normal constraint method uses a very similar approach<sup>112</sup>, but with inequality constraints to address a deficiency in the NBI method that occurs when the normal line does not cross the Pareto front. This methodology has undergone various improvements including better scaling through normalization.<sup>113</sup> A more recent improvement allows for even more efficient generation of the Pareto frontier by avoiding regions of the Pareto front where minimal tradeoffs occur.<sup>114</sup>

### 9.3.4 Evolutionary Algorithms

Gradient-free methods can, and occasionally do, use all of the above methods. However, evolutionary algorithms also enable a fundamentally different approach. Genetic algorithms, a specific type of evolutionary algorithms, were introduced in Section 7.5.\*

A genetic algorithm (GA) is amenable to an extension that can handle multiple objectives because it keeps track of a large population of designs at each iteration. If we plot two objective functions for a given population of a genetic algorithm iteration, we would get something like that shown in Fig. 9.9. The points represent the current population, and the highlighted points in the lower left are the current nondominated set. As the optimization progresses, the nondominated set moves further down and to the left and eventually converges toward the true Pareto set.

In the multiobjective version of the genetic algorithm, the reproduction and mutation phases are unchanged from the single objective version. The primary difference is in determining the fitness and the selection procedure. Here, we provide an overview of one popular approach, the NSGA-II algorithm.<sup>†</sup>

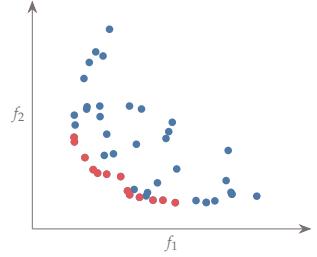
<sup>112</sup>. Ismail-Yahaya et al., *Effective generation of the Pareto frontier using the Normal Constraint method*. 2002

<sup>113</sup>. Messac et al., *Normal Constraint Method with Guarantee of Even Representation of Complete Pareto Frontier*. 2004

<sup>114</sup>. Hancock et al., *The smart normal constraint method for directly generating a smart Pareto set*. 2013

\*The first application of an evolutionary algorithm for solving a multiobjective problem was by Schaffer<sup>115</sup>.

<sup>115</sup>. Schaffer, *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms*. 1984



**Figure 9.9:** Population for a multi-objective genetic algorithm iteration plotted against two objectives. The nondominated set is highlighted at the bottom left and eventually converges toward the Pareto front.

<sup>†</sup>The NSGA-II algorithm was developed by Deb et al.<sup>101</sup> Some key developments include using the concept of domination in the selection process, preserving diversity amongst the non-dominated set, and using elitism.<sup>116</sup>

<sup>101</sup>. Deb et al., *A fast and elitist multiobjective genetic algorithm: NSGA-II*. 2002

<sup>116</sup>. Deb, *Introduction to Evolutionary Multiobjective Optimization*. 2008

A step in the algorithm is to find a nondominated set (i.e., the Pareto set), and several algorithms exist to accomplish this. In the following we use the algorithm by Kung<sup>117</sup>, which has been shown to be one of the fastest. This procedure recursively divides the population in half, finding the front for each half separately. Before calling the algorithm the population should be sorted by the first objective. First, we split the population into two halves, where the top half is superior to the bottom half in the first objective. Both populations are recursively fed back through the algorithm to find their fronts. We then initialize a merged population with the members of the top half. All members in the bottom half are checked, and any that are nondominated by any member of the top half are added to the merged population. Finally, we return the merged population as the nondominated set. The methodology is summarized in Alg. 9.5.

<sup>117</sup>. Kung et al., *On Finding the Maxima of a Set of Vectors*. 1975

---

**Algorithm 9.5:** Find nondominated set using Kung's algorithm

---

**Inputs:**

$p$ : a population sorted by the first objective

**Outputs:**

$f$ : the pareto set for the population

---

```

procedure front( $p$ )
    if length( $p$ ) = 1 then                                if there is only one point, it is the front
        return  $f$ 
    end if
    split population into two halves  $p_t$  and  $p_B$ 
    > because input was sorted,  $p_t$  will be superior to  $p_B$  in the first objective.
     $t = \text{front}(p_t)$                                 recursive call to find front for top half
     $b = \text{front}(p_B)$                                 recursive call to find front for bottom half
    initialize  $f$  with the members from  $t$                 merged population
    for  $i = 1$  to length( $b$ ) do
        dominated = false
        for  $j = 1$  to length( $t$ ) do
            if  $t_j$  dominates  $b_i$  then
                dominated = true
                break                                         no need to continue search through  $t$ 
            end if
        end for
        if not dominated then                          $b_i$  was not dominated by anything in  $T$ 
            add  $b_i$  to  $f$ 
        end if
    end for
    return  $f$ 
end procedure

```

In NSGA-II, we are interested in not just the Pareto set, but rather in ranking all members by their *dominance depth*, which is also called nondominated sorting. In this approach, all points in the population that are nondominated (i.e., the Pareto set) are given a rank of 1. Those points are then removed from the set and the next set of nondominated points is given a rank of 2, and so on (see Fig. 9.10). Note that there are alternative procedures that can perform a nondominated sorting directly, which can sometimes be more efficient, though we don't highlight them here. This algorithm is summarized in Alg. 9.6.

---

**Algorithm 9.6:** Perform nondominated sorting

---

**Inputs:** $p$ : a population**Outputs:**

rank: the rank for each member in the population

---

```

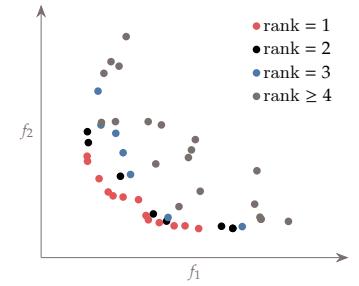
 $r = 1$                                 initialize current rank
 $s = p$                                 set sub population as entire population
while length( $s$ ) > 0 do
     $f = \text{front}(\text{sort}(s))$           identify the current front
    set rank for every member of  $f$  to  $r$ 
     $r += 1$                             increment rank
    remove all members of  $f$  from  $s$ 
end while

```

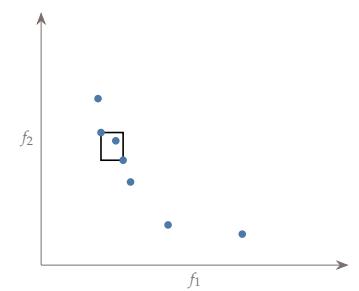
---

The new population is filled by placing all rank 1 points in the new population, then all rank 2 points, and so on. At some point, an entire group of constant rank will not fit within the new population. Points with the same rank are all equivalent as far as Pareto optimality is concerned, so an additional sorting mechanism is needed to determine which members of this group to include.

The way that we perform selection within a group that can only partially fit is to try to preserve diversity as much as possible. Points in this last group are ordered by their *crowding distance*, which is a measure of how spread apart the points are. The algorithm seeks to preserve points that are well spread. For each point, a hypercube in objective space is formed around it, which in NSGA-II is referred to as a *cuboid*. Figure 9.11 shows an example cuboid considering the rank 3 points from Fig. 9.10. The hypercube extends to the function values of its nearest neighbors in function space. That does not mean that it



**Figure 9.10:** Points in the population highlighted by rank.



**Figure 9.11:** A cuboid around one point demonstrating the definition of crowding distance (except that the distances are normalized).

necessarily touches its neighbors as the two closest neighbors can differ for each objective. The sum of the dimensions of this hypercube is the crowding distance. When summing the dimensions, each dimension is normalized by the maximum range of that objective value. For example, considering only  $f_1$  for the moment, if the objectives were in ascending order, then the contribution of point  $i$  to the crowding distance would be:

$$d_{1,i} = \frac{f_{1,i+1} - f_{1,i-1}}{f_{1,N} - f_{1,1}} \quad (9.16)$$

Sometimes, instead of using the first and last points in the current objective set, user-supplied values are used for the min and max values of  $f$  that appear in that denominator. The anchor points (the single objective optima) are assigned a crowding distance of infinity as we want to preference their inclusion. The algorithm for crowding distance is shown in Alg. 9.7.

---

**Algorithm 9.7: Crowding distance**


---

**Inputs:** $p$ : a population**Outputs:** $d$ : crowding distances

---

```

initialize d with zeros
for  $i = 1$  to number of objectives do
    set  $f$  as a vector containing the  $i^{\text{th}}$  objective for each member in  $p$ 
     $s = \text{sort}(f)$  and let  $I$  contain the corresponding indices ( $s = f_I$ )
     $d_{I_1} = \infty$  anchor points receive an infinite crowding distance
     $d_{I_n} = \infty$ 
    for  $j = 2$  to  $\text{length}(p) - 1$  do add distance for interior points
         $d_{I_j} += (s_{j+1} - s_{j-1}) / (s_N - s_1)$ 
    end for
end for
return d

```

---

We can now put together the pieces in the overall algorithm. The crossover and mutation operations remain the same. Tournament selection (Fig. 7.17) is modified slightly to use the ranking and crowding metrics of this algorithm. In the tournament, a member with a lower rank is superior. If two members have the same rank, then the one with the larger crowding distance is selected. This procedure is called *crowded tournament selection*. After reproduction/mutation, instead of replacing the parent generation with the offspring generation, both the

parent generation and offspring generation are saved as candidates for the next generation. This preserves *elitism*, which means that the best member in the population is guaranteed to survive. The population size is now twice its original size ( $2N$ ) and the selection process must reduce the population back down to size  $N$ . This is done using the procedure explained previously. The new population is filled by including all rank 1 members, rank 2 members, etc., until an entire rank can no longer fit. Inclusion for members of that last rank are done in order of largest crowding distance until the population is filled. The general algorithm is summarized in Alg. 9.8. Note that many variations are possible, so while the algorithm is based on the concepts of NSGA-II, the details may differ somewhat.

---

**Algorithm 9.8:** Elitist non-dominated sorting genetic algorithm
 

---

**Inputs:** $\bar{x}$ : Variable upper bounds $\underline{x}$ : Variable lower bounds $f(x)$ : function**Outputs:** $x^*$ : Optimal point

---

 Generate initial population

**while** Stopping criterion is not satisfied **do**

 Using a parent population  $P$ , evaluate fitness, perform selection, crossover, mutation as in a standard GA to produce an offspring population  $O$ , except modify selection to use a crowded tournament selection
 $C = P \cup O$ *combine populations*Compute  $rank_i$  for  $i = 1, 2, \dots$  of  $C$  using Alg. 9.6*> Fill new parent population with as many whole ranks as possible* $P = \emptyset$  $r = 1$ **while** true **do**  set  $F$  as all  $C_i$  with  $rank_i = r$   **if** length( $P$ ) + length( $F$ ) >  $N_p$  **then**    **break**  **end if**  add  $F$  to  $P$    $r += 1$ **end while***> For last rank that doesn't fit, add by crowding distance***if** length( $P$ ) <  $N_p$  **then***population isn't full*   $d = \text{crowding}(F)$ *Alg. 9.7, using last  $F$  from terminated loop above*   $m = N_p - \text{length}(P)$ *determine how many members to add*  sort  $F$  by the crowding distance  $d$  in descending order

```

    add the first  $m$  entries from  $F$  to  $P$ 
end if
end while

```

---

**Example 9.9:** Filling a new population in NSGA-II

After reproduction and mutation, we are left with a combined population of parents and offspring. In this small example the combined population is of size 12 and so we must reduce it back to 6. This example has two objectives, and the values for each member in the population is shown below. To refer to these members, just for this purpose of this example, we have assigned each member with a lettered label. The population is plotted in Fig. 9.12.

	A	B	C	D	E	F	G	H	I	J	K	L
$f_1$	5	7	10	1	3	10	5	6	9	6	9	4
$f_2$	8	9	4	4	7	6	10	3	5	1	2	10

First, we compute the ranks using Alg. 9.6, resulting in the following output:

A	B	C	D	E	F	G	H	I	J	K	L
3	4	3	1	2	4	4	2	3	1	2	3

We see that current nondominated set consists of points D and J and that there are four different ranks.

Next, we start filling the new population in order of rank. Our max capacity is 6, so all of rank 1 (D, J) and rank 2 (E, H, K) fit. We cannot add rank 3 (A, C, I, L) because the population size would be 9. So far our new population consists of [D, J, E, H, K]. To choose which items from rank 3 continue forward, we compute the crowding distance for the members of rank 3:

A	C	I	L
1.67	$\infty$	1.5	$\infty$

We would then add these in order: C, L, A, I, but only have room for one, so we add C and complete this iteration with a new population of [D, J, E, H, K, C].

---

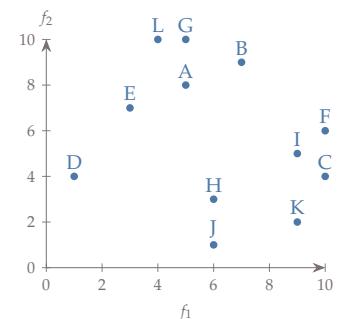


Figure 9.12: Population for Ex. 9.9

The main advantage of this multiobjective approach is that if an evolutionary algorithm is appropriate for solving a given single objective problem, then the extra information needed for a multiobjective problem is already there and therefore solving the multiobjective problem does not incur an additional computational cost. The pros and cons of this approach as compared to the previous approaches are basically the pros and cons of gradient-based versus gradient-free methods with the exception that the multiobjective gradient-based approaches require solving multiple problems to generate the Pareto front. Still, solving

multiple gradient-based problems is often more efficient than solving one gradient-free problem, especially for problems with a large number of design variables.

## 9.4 Summary

Multiobjective optimization is particularly useful in quantifying tradeoff sensitivities between critical metrics. It is also useful when a family of potential solutions is sought, rather than a single solution. Some scenarios where a family of solutions might be preferred is when the models used in optimization are low fidelity and higher fidelity design tools will be applied, or when more investigation is needed and only candidate solutions are desired at this stage.

The presence of multiple objectives changes what it means for a design to be optimal. The concept of Pareto optimality was introduced where a design is nondominated by any other design. The weighted sum method is perhaps the most well-known approach, but it is not recommended as other methods are just as easy and much more efficient. The constraint epsilon method is still simple, but almost always preferable to the weighted sum method. If willing to use a more complex approach, the normal boundary intersection method is even more efficient at capturing a Pareto front.

Some gradient-free methods are also effective at generating Pareto fronts, particularly a multiobjective genetic algorithm. While gradient-free methods are sometimes associated with multiobjective problems, gradient-based algorithms may be the more effective approach for many multiobjective problems.

## Problems

### 9.1 Answer *true* or *false* and justify your answer.

- a) The solution of multiobjective optimization problems is usually an infinite number of points.
- b) It is advisable to include as many objectives as you can in your problem formulation to make sure you get the best possible design.
- c) Multiobjective optimization allows us to quantify tradeoffs between objectives and constraints.
- d) If the objectives are separable, that means that they can be minimized independently and that there is no Pareto front.

- e) A point  $A$  dominates point  $B$  if it is better than  $B$  in at least one objective.
- f) The Pareto front is the set of all the points that dominate all other points in the objective space.
- g) When a point is Pareto optimal, you cannot make either of the objectives better.
- h) The weighted-sum method obtains the Pareto front by solving optimization problems with different objective functions.
- i) The  $\epsilon$ -constraints method obtains the Pareto front by minimizing each objective in turn while constraining the others.
- j) The utopia point is the point where every objective has a minimum value.
- k) It is not possible to compute a Pareto front with a single-objective optimizer.
- l) Because GAs optimize by evolving a population of diverse designs, they can be used for multiobjective optimization without modification.

9.2 Which of the following function value pairs would be Pareto optimal in a multiobjective minimization problem (may be more than one)?

- (20, 4)
- (18, 5)
- (34, 2)
- (19, 6)

9.3 You seek to minimize the following two objectives:

$$\begin{aligned}f_1(x) &= x_1^2 + x_2^2 \\f_2(x) &= (x_1 - 1)^2 + 20(x_2 - 2)^2\end{aligned}$$

Identify the Pareto front using the weighted sum method with 11 evenly spaced weights: 0, 0.1, 0.2, ..., 1. If some parts of the front are underresolved, discuss how you might select weights for additional points.

9.4 Repeat Prob. 9.3 with the epsilon-constraint method. Constrain  $f_1$  with 11 evenly spaced points between the anchor points. Contrast the Pareto front with that of the previous problem, and discuss whether improving the front with additional points will be easier with the previous method, or with this method.

- 9.5 Repeat Prob. 9.3 with the normal boundary intersection method using the following 11 evenly spaced points:  $b = [0, 1], [0.1, 0.9], [0.2, 0.8], \dots, [1, 0]$
- 9.6 Consider a two-objective population with the following combined parent/offspring population (objective values shown for all sixteen members).

[6.0, 8.0]  
[6.0, 4.0]  
[5.0, 6.0]  
[2.0, 8.0]  
[10.0, 5.0]  
[6.0, 0.5]  
[8.0, 3.0]  
[4.0, 9.0]  
[9.0, 7.0]  
[8.0, 6.0]  
[3.0, 1.0]  
[7.0, 9.0]  
[1.0, 2.0]  
[3.0, 7.0]  
[1.5, 1.5]  
[4.0, 6.5]

Develop code based on the NSGA-II procedure and determine the new population at the end of this iteration. Detail the results of each step during the process.

A surrogate model, also known as a response surface model or meta-model, is an approximate model of a functional output that represents a “curve fit” to some underlying data. The goal is to create a surrogate that is much faster to compute than the original function, but that still retains sufficient accuracy away from known data points. The surrogate model is also usually smoother than the original function.

By the end of this chapter you should be able to:

1. Identify and describe the steps in surrogate-based optimization.
2. Understand and use Latin hypercube sampling.
3. Select optimized parameters for a given surrogate model.
4. Perform cross validation as part of model selection.
5. Describe two approaches to infill.

## 10.1 When to Use a Surrogate

There are various scenarios for which a surrogate model might be useful:

- When the simulation is expensive to evaluate and you expect to evaluate it many times.
- When the simulation is noisy.
- When the model is derived from experimental data (which may be both expensive and noisy).
- When you want to better understand functional relationships between the inputs and outputs.

- When multiple model fidelities are involved and a surrogate may be able to provide a correction between fidelities.

Our interest is not just in building surrogate models, but rather in performing optimization using a surrogate model. This topic is called *surrogate-based optimization* (SBO). SBO is a rich subfield and this chapter presents only a brief introduction.

A broad overview of the steps in a SBO is shown in Fig. 10.1. First, sampling methods are used to choose the initial points to evaluate the function, or conduct experiments at. These points are sometimes referred to as *training data*. Next, a surrogate model is created from the sampled points. We then perform an optimization using the surrogate model. Based on the results of the optimization we can include additional points in the sample and reconstruct the surrogate (infill). This process is repeated until some convergence criteria or maximum number of iterations is reached. The optimization step reuses the techniques we have already discussed. The new steps we discuss in this chapter are sampling, constructing a surrogate, and infill.\* In some procedures infill is omitted, the surrogate is fully constructed upfront and not subsequently updated. Many of the concepts discussed in this chapter are of broader usefulness in optimization beyond just SBO.

## 10.2 Sampling

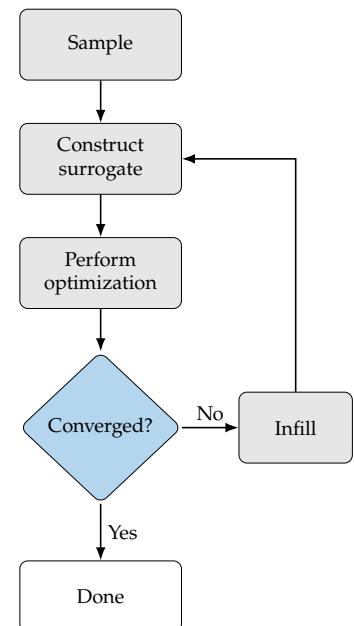
Sampling methods select the evaluation points for constructing the initial surrogate. These evaluation points must be chosen carefully.

**Example 10.1:** Full grid sampling is not scalable.

Imagine a simulation model computing the endurance of an aircraft. Assume that one simulation takes a few hours on a supercomputer and so evaluating many points is prohibitive. If we only wanted to understand how endurance varied with one variable, like wing span, we could run the simulation 10 times and likely create a fairly useful curve that could predict endurance at wing spans we didn't directly evaluate. Now imagine that we have nine additional input variables that we want to use: wing area, taper ratio, wing root twist, wing tip twist, wing dihedral, propeller spanwise position, battery size, tail area, and tail longitudinal position. If we discretized all ten variables with ten intervals each, we would need to run  $10^{10}$  simulations in order to assess all combinations. With 3 hours per simulation, that would take almost 5 million years!

\*Forrester *et al.*<sup>118</sup> provides a nice introduction to this topic with much more depth than can be provided in this chapter.

<sup>118</sup> Forrester *et al.*, *Engineering Design via Surrogate Modelling: A Practical Guide*, 2008



Ex. 10.1 highlights one of the major challenges of sampling methods: the curse of dimensionality. For SBO, using a large number of variables

**Figure 10.1:** Overview of surrogate-based optimization procedure.

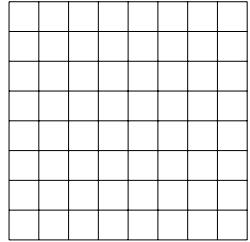
is costly, and so we need to identify the most important, or most influential variables. There are many methods that can help us do that, but they are beyond the scope of this introductory text. Instead, we assume that the most influential variables have already been determined so that the dimensionality is reasonable. However, even with a modest number of variables, a *full grid search* is highly inefficient. We are interested in sampling methods that can efficiently characterize our design space of interest. In this introduction we focus on a popular sampling method called *Latin hypercube sampling* (LHS).

LHS is a random process, but it is much more effective and efficient than pure random sampling. In random sampling each sample is independent of past samples, but in LHS we choose all samples before hand in order to represent the variability effectively. Consider two random variables with some bounds, whose design space we could represent as a square. Say we wanted only eight samples, we could divide the design space into eight intervals in each dimension as shown in Fig. 10.2.

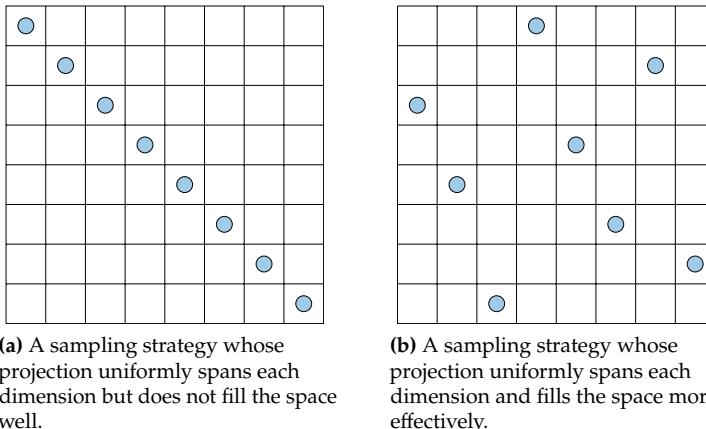
A full grid search would identify a point in every little box, but this does not scale well. To be as efficient as possible, and still cover the variation, we would want each row and each column to have one sample in it. In other words, the projection of points onto each dimension should be uniform. This is called a Latin square and the generalization to higher dimensions is a Latin hypercube. There are a large number of possible ways we could achieve this, and some choices will be better than others. Consider the sampling plan shown in Fig. 10.3a. This plan achieves our criteria but clearly does not fill the space and likely will not capture the relationships between design parameters well. Alternatively Fig. 10.3b has a sample in each row and column while also spanning the space much more effectively.

As it turns out, LHS is itself an optimization problem. It seeks to maximize the distance between the samples with the constraint the projection on each axis follow a chosen probability distribution (usually uniform as in the above examples, but it could also be something else like Gaussian). There are many possible solutions, and so some randomness is involved. Rather than relying on the law of large numbers to fill out our chosen probability distributions, we enforce it as a constraint. This method still generally requires a large number of samples to accurately characterize the design space, but usually far less than pure random sampling.

Most scientific packages include convenience methods for random sampling from typical distributions (e.g., uniform or normal), but you can also easily sample from an arbitrary distribution using a technique

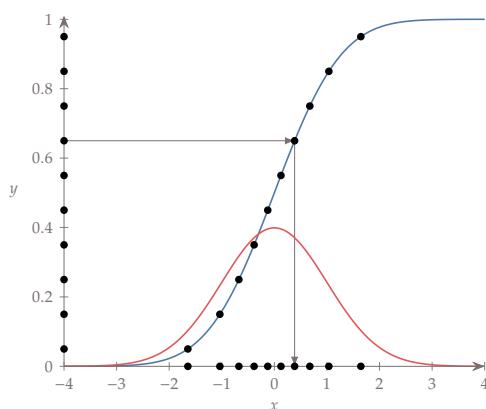


**Figure 10.2:** A two-dimensional design space divided into 8 intervals in each dimension.



**Figure 10.3:** Contrasting sampling strategies that both fulfill the uniform projection requirement.

called *inversion sampling*. Assume that we want to generate samples  $x$  from an arbitrary PDF  $p(x)$  or equivalently from the corresponding CDF  $P(x)$ . The probability integral transform states that for any continuous CDF:  $y = P(x)$  the variable  $y$  is uniformly distributed (a simple proof, but not shown here to avoid introducing additional notation). The procedure is to randomly sample from a uniform distribution (e.g., generate  $y$ ), then compute the corresponding  $x$  such that  $P(x) = y$ . This latter step is known as an inverse CDF, a percent-point function, or a quantile function and this process is depicted pictorially in Fig. 10.4 for a normal distribution. Note that this same procedure allows you to use LHS with any distribution, simply by generating the samples on a uniform distribution.



**Figure 10.4:** An example of inversion sampling with a normal distribution. A few uniform samples are shown on the y-axis. The points are evaluated by the inverse CDF, depicted by the arrows passing through the CDF for a normal distribution (blue curve). If enough samples are drawn, the resulting distribution will be the PDF of a normal distribution (red curve).

In addition to use in SBO, LHS is also very useful in other applications discussed in this book including: initializing a Genetic Algorithm

(Section 7.5), initializing a Particle Swarm Method (Section 7.6), performing restarts in a gradient-based method for exploration of multiple minima (Tip 4.24), or choosing the points to run in a Monte Carlo simulation (Section 12.5.2).

### 10.3 Constructing a Surrogate

Many types of surrogate models are possible, some physics-based, others mathematically-based, and some that are a hybrid (particularly with multi-fidelity models). We will discuss the fundamentals of a simple and frequently used mathematically-based model: linear regression. A linear regression model does not mean that the surrogate is linear, but that the model is linear in its coefficients (i.e., linear in the parameters we are estimating).

In this section, we discuss two linear regression models: polynomial models and Kriging. Many others exist, but these are chosen because of their popularity and because they will illustrate many of the important concepts in surrogate models.

A general linear regression model looks like:

$$\hat{f} = w^T \psi \quad (10.1)$$

where  $w$  is a vector of weights and  $\psi$  is a vector of basis functions. Another popular regression model is Kriging, where the basis functions are:

$$\psi^{(i)} = \exp\left(-\sum_j \theta_j |x_j^{(i)} - x_j|^{p_j}\right) \quad (10.2)$$

In general, the basis functions can be any set of functions that we choose, though often we would like these functions to be orthogonal.

**Example 10.2:** Data fitting can be posed as a linear regression model.

Consider a simple quadratic fit:  $\hat{f} = ax^2 + bx + c$ . This is a linear regression model because it is linear in the coefficients we wish to estimate:  $w = [a, b, c]$ . The basis functions would be  $\psi = [x^2, x, 1]$ . For a more general n-dimensional polynomial model, the basis function would be polynomials like:

$$\psi \in \{1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2^2, x_1^2x_3, \dots\} \quad (10.3)$$

To construct a linear regression model there are two tasks: 1) determine what terms to include in  $\psi$ , and 2) estimate  $w$  to minimize some error. For a given set of  $\psi$  terms, the latter is straightforward.

From sampling, we already selected a bunch of evaluation points and computed their corresponding function values. We call these values the training data:  $(x^{(i)}, f^{(i)})$ . We want to choose the weights  $w$  to minimize the error between our predicted function values  $\hat{f}$  and the actual function values  $f^{(i)}$ . Errors can be positive or negative but of course any error is bad, so what we want to minimize is not the sum of the errors, but rather the sum of the square of the errors<sup>†</sup>:

$$\text{minimize} \sum_i (\hat{f}(x^{(i)}) - f^{(i)})^2 \quad (10.4)$$

The solution to this optimization problem is called a least squares solution as discussed in Section 11.2. Let us rewrite  $\hat{f}$  in matrix form so that it is compact for all  $x^{(i)}$ :

$$\hat{f} = \Psi w \quad (10.5)$$

where  $\Psi$  is matrix:

$$\Psi = \begin{bmatrix} \vdash & \psi(x^{(1)})^T & \vdash \\ \vdash & \psi(x^{(2)})^T & \vdash \\ \vdots & & \\ \vdash & \psi(x^{(n)})^T & \vdash \end{bmatrix} \quad (10.6)$$

Thus, the same minimization problem can be expressed as:

$$\text{minimize} ||\Psi w - f||^2 \quad (10.7)$$

where

$$f = \begin{bmatrix} f^{(1)} \\ f^{(2)} \\ \vdots \\ f^{(n)} \end{bmatrix} \quad (10.8)$$

The matrix  $\Psi$  is of size  $(m \times n)$  where  $m > n$ . This means that there should be more equations than unknowns, or that we have sampled more points than the number of polynomial coefficients we need to estimate. This should make sense because our polynomial function is only an assumed form, and generally not an exact fit to the actual underlying function. Thus, we need more data to create a reasonable fit.

This is exactly the same problem as  $y = Ax$  where  $A \in \mathcal{R}^{m \times n}$ . There are more equations than unknowns so generally there is not a solution (the problem is called overdetermined). Instead, we seek the solution that minimizes the error  $||Ax - y||^2$ .

<sup>†</sup>It is not arbitrary that we minimize the sum of the squares rather than the sum of the absolute values or some other metric. If we assume that the error in our linear regression model is independently distributed according to a Gaussian distribution (which is a typical assumption in the absence of other information), and wish to find the  $w$  that maximizes the probability that we would observe the data  $\hat{f}$ , then the resulting optimization problem reduces to a sum of the square of the errors.

---

Tip 10.3: Least squares is not the same as a linear system solution.

In Julia or Matlab you can solve this with  $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$ , but keep in mind that for  $A \in \mathcal{R}^{m \times n}$  this syntax performs a least squares solution, not a linear system solution because it would for a full rank  $n \times n$  system. This overloading is generally not used in other languages, for example in Python rather than using `numpy.linalg.solve` you would use `numpy.linalg.lstsq`.

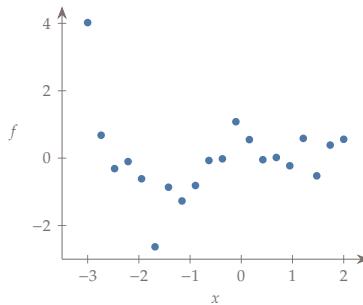
The other important consideration for developing our surrogate model is choosing the basis functions in  $\psi$ . Sometimes you may know something about the model behavior and thus what type of basis functions should be used, but generally the best way to determine the basis functions is through cross validation. Cross validation is also useful in characterizing error, even if we already have a chosen set of basis functions.

**Example 10.4:** The dangers of overfitting.

Consider a simple underlying function with Gaussian noise added to simulate experimental or noisy computational data.

$$y = 0.2x^4 + 0.2x^3 - 0.9x^2 + \mathcal{N}(0, \sigma), \text{ where } \sigma = 0.75 \quad (10.9)$$

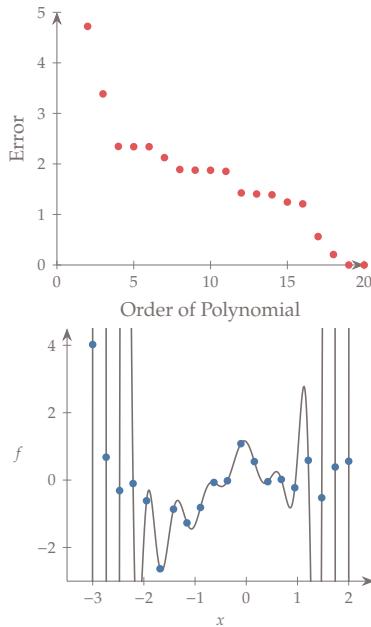
We will create the data at 20 points in the interval  $[-3, 2]$  (see Fig. 10.5).



**Figure 10.5:** Data from a numerical model or experiments.

For a real problem we do not know the underlying function and the dimensionality is often too high to visualize. Determining the right basis functions to use can be difficult. If we are using a polynomial basis we might try to determine the order by trying each case (e.g., quadratic, cubic, quartic, etc.) and measuring the error in our fit (Fig. 10.6).

It seems as if the higher the order of the polynomial, then the lower the error. For example, a 20th order polynomial reduces the error to almost zero. The problem is that while the error may be low on this set of data, we expect the predictive capability of such a model for future data points to be poor. For example, Fig. 10.7 shows a 19th order fit to the data. The model passes right through the points, but its predictive capability is poor.



**Figure 10.6:** Error in fitting data with different order polynomials.

**Figure 10.7:** A 19th order polynomial fit to the data. Low error, but poor predictive ability.

This is called *overfitting*. Of course, we also want to avoid the opposite problem of underfitting where we do not have enough degrees of freedom to create a useful model (e.g., think of a linear fit for the above example).

The solution to the overfitting problem highlighted in Ex. 10.4 is cross validation. *Cross validation* means that we use one set of data for training (creating the model), and a different set of data for assessing its predictive error. There are many different ways to perform cross-validation, we will describe two. *Simple cross validation* consists of the following steps:

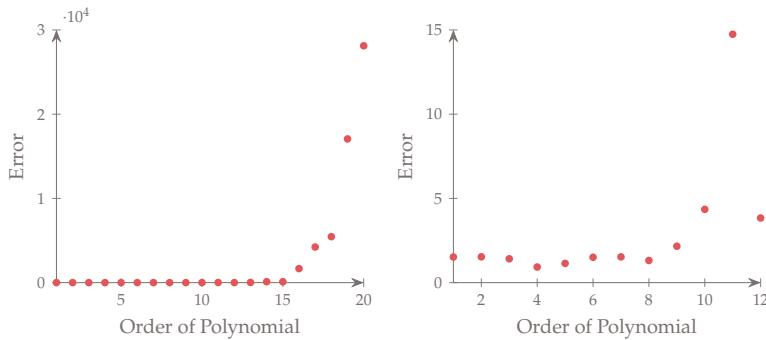
1. Randomly split your data into a training set and a validation set (e.g., a 70/30 split).
2. Train each candidate model (the different options for  $\psi$ ) using only the training set, but evaluate the error with the validation set.
3. Choose the model with the lowest error on the validation set, and optionally retrain that model using all of the data.

An alternative option may be useful if you have few data points and so can't afford to leave much out for validation. This method is called *k-fold cross validation* and while it is more computationally intensive, it makes better use of all of your data:

1. Randomly split your data into  $n$  sets (e.g.,  $n = 10$ ).
2. Train each candidate model using the data from all sets except one (e.g., 9 of the 10 sets), and use the remaining set for validation. Repeat for all  $n$  possible validation sets and average your performance.
3. Choose the model with the lowest average error on all the  $n$  validation sets.

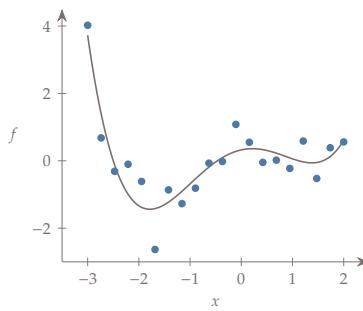
**Example 10.5:** Cross validation is used to avoid overfitting.

This example continues from Ex. 10.4. First, we perform k-fold cross-validation using ten divisions. The average error across the divisions using the training data is shown in Fig. 10.8.



**Figure 10.8:** Error from k-fold cross validation.

The error becomes extremely large as the polynomial order becomes large. Zooming in on the flat region we see a range of options with similar errors. Amongst similar solutions, one generally prefers the simplest model. In this case, a fourth-order polynomial seems reasonable. A fourth-order polynomial is compared against the data in Fig. 10.9. This model has much better predictive capability.



**Figure 10.9:** A 4th order polynomial fit to the data.

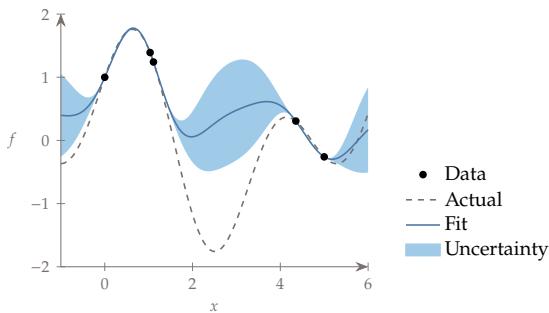
An important class of linear regression models are those that use radial basis functions. A radial basis function is a basis function that depends on distance from some center point:

$$\psi^{(i)} = \psi\left(||x - c^{(i)}||\right), \quad (10.10)$$

of which one of the more popular models is Kriging:

$$\psi^{(i)} = \exp\left(-\sum_j \theta_j |x_j^{(i)} - x_j|^{p_j}\right) \quad (10.11)$$

A Kriging basis is a generalization of a Gaussian basis (which would have  $\theta = 1/\sigma^2$  and  $p = 2$ ). These types of models are useful because in addition to creating a model they also predict the uncertainty in the model through the surrogate. An example of this is shown in Fig. 10.10. Notice how the uncertainty goes to zero at the known data points, and becomes largest when far from known data points.



**Figure 10.10:** A Kriging fit to the input data (circles) and a shaded confidence interval.

The key advantage of a Gaussian process model is this ability to predict not just a surrogate function but also its uncertainty. Another advantage is its flexibility. Unlike the polynomial models, we make fewer assumptions upfront. This flexibility is also a disadvantage in that we generally need many more function calls to produce a reasonable model. The other main disadvantage of Gaussian process models is that they tend to introduce lots of local minima.

---

**Tip 10.6:** Surrogate modeling toolbox.

The surrogate modeling toolbox<sup>‡</sup> is a useful package for surrogate modeling with a particular focus on providing derivatives for use in gradient-based optimization.

---

<sup>‡</sup><https://smt.readthedocs.io/>

## 10.4 Infill

There are two main approaches to infill: prediction-based exploitation and error-based exploration. Typically, only one infill point is chosen at a time with the assumption that evaluating the model is computationally expensive, but recreating and evaluating the surrogate is cheap.

For the polynomial models discussed in the previous section the only real option is exploitation. A prediction-based exploitation infill strategy adds an infill point wherever the surrogate predicts the optimum. The reasoning behind this method, is that in SBO we do not necessarily care about having a globally accurate surrogate, but rather only care about having an accurate surrogate near the optimum. The most reasonable point to sample at is the optimum predicted by the surrogate. Likely, the location predicted by the surrogate will not be at the true optimum, but it will add valuable information as we recreate the surrogate and reoptimize, repeating the process until convergence. This approach generally allows for the quickest convergence to an optimum. Its downside is that for problems with multiple local optima we are likely to converge prematurely to an inferior local optimum.

An alternative approach is called error-based exploration. This approach requires the use of a Gaussian process model (mentioned in the previous section) that not only predicts function values, but also uncertainties. In exploration we may not want to just sample where the mean is low (this is the same as exploitation), but we also do not want to just sample where the error is high (this is essentially just a larger sampling plan). Instead, we want to sample where the probability of finding improvement is highest. One metric with this intent is called *expected improvement*.

Let the best solution we have found so far be called  $x^*$ , and  $f(x)$  is our objective function. The improvement for any new test point  $x$  is then given by:

$$I(x) = \max(f(x^*) - f(x), 0) \quad (10.12)$$

In other words, if  $f(x) \geq f(x^*)$  there is no improvement but if  $f(x) < f(x^*)$  then the improvement is just the magnitude of the objective decrease. However, we need to keep in mind that  $f(x)$  is not a deterministic value in this model, but rather a probability distribution. Thus, the expected improvement is the expected value (or mean) of the improvement

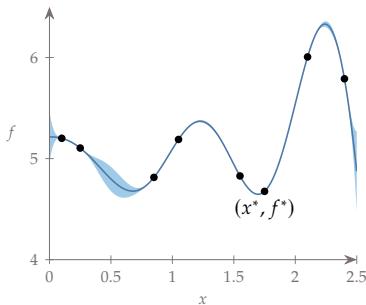
$$EI(x) = \mathbb{E} [\max(f(x^*) - f(x), 0)] . \quad (10.13)$$

For a Gaussian process model, the expected value can be determined analytically. The selected infill point is the point with the highest

expected improvement. After sampling, we recreate the surrogate and repeat.

**Example 10.7:** Expected improvement.

Consider the one-dimensional function with data points and fit shown in Fig. 10.11. § The best point we have found so far is denoted in the figure as  $x^*, f^*$ . For a Gaussian process model, the fit also provides an uncertainty distribution as shown in the shaded region.

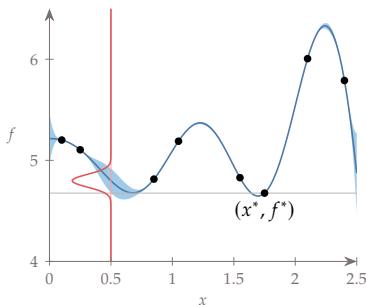


§This data is based on an example from Rajnarayan *et al.*<sup>119</sup>.

119. Rajnarayan *et al.*, *A Multifidelity Gradient-Free Optimization Method and Application to Aerodynamic Design*. 2008

**Figure 10.11:** A one-dimensional function with a Gaussian process model surrogate fit and uncertainty.

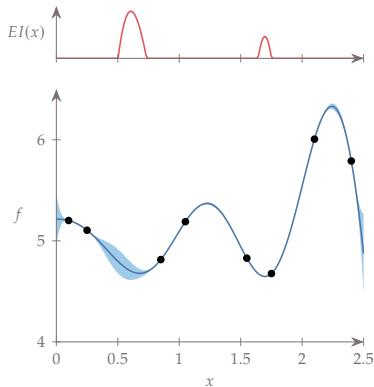
Now imagine we want to evaluate this function at some new test point  $x_{\text{test}} = 0.5$ . In Fig. 10.12 the probability distribution for the objective at  $x_{\text{test}}$  is shown in red (imagine that the probability distribution was coming out of the page). The shaded blue region is the probability of improvement over the best point. Expected value is similar to the probability of improvement but rather than return a probability it returns the magnitude of improvement expected. That magnitude may be more helpful in defining a stopping criteria as opposed to a probability.



**Figure 10.12:** At a given test point ( $x_{\text{test}} = 0.5$ ) we highlight the probability distribution and the expected improvement in the shaded blue region.

Now, let us evaluate the expected improvement not just at  $x_{\text{test}} = 0.5$  but across the domain. The result is shown by the red function in Fig. 10.13. The spike on the right tells us that we expect improvement by sampling close to our best known point, but the expected improvement is rather small. The spike on the left tells us that there is a promising region where the surrogate suggests a relatively high potential improvement. Notice that the metric does not simply

capture regions with high uncertainty, but rather regions with high uncertainty in areas that are likely to lead to improvement. For our next sample, we would choose the location with the highest expected improvement, recreate the fit and repeat.



**Figure 10.13:** The process is repeated by evaluating expected improvement across the domain.

## 10.5 Deep Neural Networks

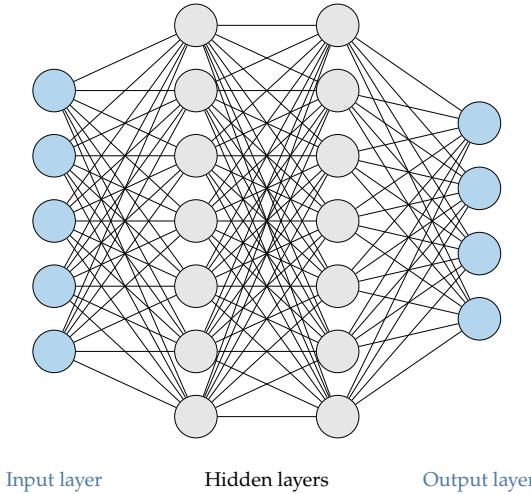
Neural networks loosely mimic the brain, which consists of a vast network of neurons. In neural networks, each neuron is a node that represents a simple function. A network defines chains of these simple functions to obtain composite functions that are much more complex. For example, three simple functions:  $f^{(1)}, f^{(2)}, f^{(3)}$  may be chained into the composite function (or network):

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x))) \quad (10.14)$$

Even though each individual function may be simple, the composite function can exhibit complex behavior. Most deep neural networks are *feedforward* networks, meaning information flows from inputs  $x$  to outputs  $f$ . Recurrent neural networks include feedback connections.

Figure 10.14 shows a diagram of a neural network. Each node represents a neuron, and these neurons are connected between consecutive layers forming a dense network. The first layer is called the *input layer*, the last is called the *output layer*, and the middle layer(s) are called *hidden layers*. The total number of layers is called the network's *depth*. The usage of the phrase deep neural networks instead of just neural networks or artificial neural networks reflects recent advances that have allowed researchers to train networks that are much deeper than was

possible before. The phrase neural is used because these models are inspired by neurons in a brain.



**Figure 10.14:** A representation of a small neural net.

The first and last layers are the inputs and outputs of our surrogate model. Each neuron in the hidden layer represents a function. This means that the output from a neuron is a number, and thus the output from a whole layer can be represented as a vector  $x$ . We call  $x^{(k)}$  the vector of values for layer  $k$ , and  $x_i^{(k)}$  is the value for the  $i$ th neuron in layer  $k$ . Let us consider just one neuron in layer  $k$ . This neuron is connected to many neurons from the previous layer  $k-1$  (see first part of Fig. 10.15). We need to choose a functional form for this neuron taking in the values from the previous layer as inputs. A linear function is too simple. Chaining together linear functions will only result in a linear composite function, so the function for this neuron must be nonlinear. The most common choice for hidden layers is a linear function passed through a second activation function that creates the nonlinearity. Let us first focus on the linear portion, which produces an intermediate variable we call  $z$  (figure):

$$z = \sum_{j=1}^n w_j x_j^{(k-1)} + b \quad (10.15)$$

or in vector form:

$$z = w^T x^{(k-1)} + b \quad (10.16)$$

Notice that the first term is just a weighted sum of the values from the neurons in the previous layer. The  $w$  vector contains the weights. The  $b$  term is called the bias, which provides an offset allowing us to scale

the significance of the overall output. This process is summarized in the second column of Fig. 10.15.

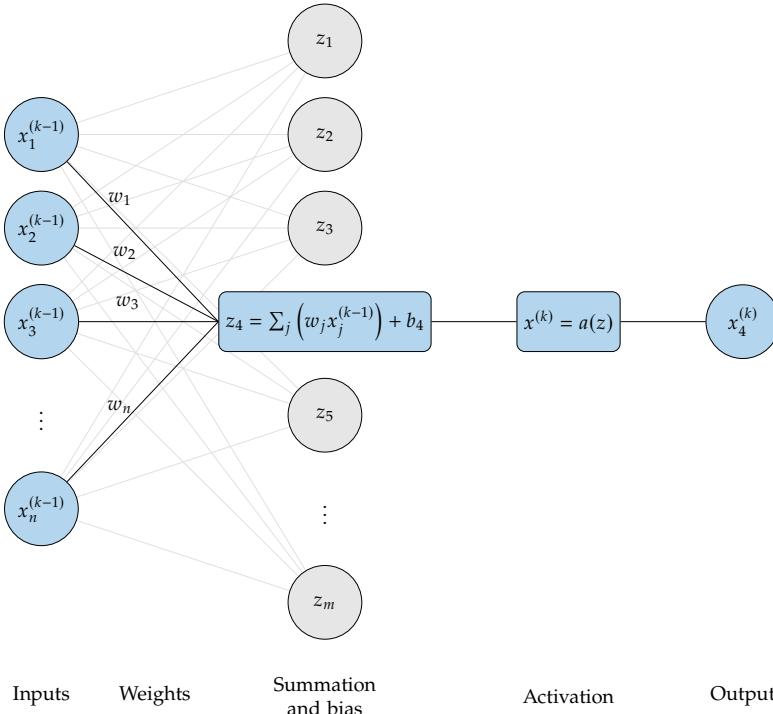


Figure 10.15: Typical functional form for a neuron in the neural net.

Next, we pass  $z$  through an *activation function*, which we will call  $a(z)$ . Historically, a sigmoid function (top of Fig. 10.16) was almost always used as the activation function:

$$a(z) = \frac{1}{1 + e^{-z}} \quad (10.17)$$

Notice that this function produces values between zero and one, so large negative values would become insignificant (close to zero) and large positive values would produce results close to one. Most modern neural nets now use a rectified linear unit (ReLU) as the activation function (bottom of Fig. 10.16):

$$a(z) = \max(0, z) \quad (10.18)$$

The ReLU has been found to be far more effective in producing accurate neural nets. Notice that this activation function completely eliminates negative inputs. Thus, we see that the bias term can be thought of as a threshold establishing what constitutes a significant value. This last step is summarized in the final two columns of Fig. 10.15.

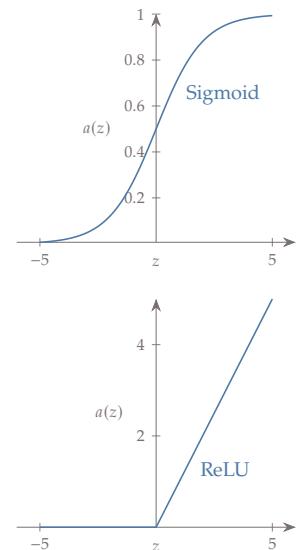


Figure 10.16: Activation functions.

Combining the linear function with the activation function produces the output for this  $i^{\text{th}}$  neuron:

$$x_i^{(k)} = a(w^T x^{(k-1)} + b_i) \quad (10.19)$$

To compute across all the neurons in this layer, the weights  $w$  for this one neuron would form one row in a matrix of weights  $W$ .

$$\begin{bmatrix} x_1^{(k)} \\ \vdots \\ x_i^{(k)} \\ \vdots \\ x_{n_k}^{(k)} \end{bmatrix} = a \left( \begin{bmatrix} & \dots & & x_1^{(k-1)} \\ & \vdots & & \vdots \\ W_{i1} & \dots & W_{ij} & \dots & W_{i,n_{k-1}} \\ & \vdots & & \vdots & \\ & \dots & & x_j^{(k-1)} & \\ & & & \vdots & \\ & & & x_{n_{k-1}}^{(k-1)} & \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{n_k} \end{bmatrix} \right) \quad (10.20)$$

or

$$x^{(k)} = a(W x^{(k-1)} + b) \quad (10.21)$$

The activation function is applied separately for each row. The below equation is more explicit (where  $w_i$  is the  $i^{\text{th}}$  row of  $W$ ), though we generally use the above equation as shorthand.

$$x_i^{(k)} = a(w_i^T x^{(k-1)} + b_i) \quad (10.22)$$

Our neural net is now parameterized by a bunch of weights and biases, and we need to determine the optimal value for these parameters (e.g., train the network) by supplying a large set of training data. In the example of Fig. 10.14 there is a layer of 5 neurons, 7 neurons, 7 neurons, then 4 neurons and so there would be  $5 \times 7 + 7 \times 7 + 7 \times 4$  weights and  $7 + 7 + 4$  bias terms giving a total of 130 design variables. This represents a very small neural net as there are few inputs and few outputs. Large neural nets can have millions of variables. We need to optimize those design variables to minimize a cost function.

The cost function combines the results from the output layer into a single objective. We will call the inputs to the neural net  $x$ , the outputs from the neural net  $f(x)$ , and use  $y$  to represent the training data. Ideally, we want to adjust the parameters such that  $f(x)$  closely matches the data  $y$ . For example,  $x$  could be parameters that set the shape of a structure, and the outputs  $y$  the stresses at various locations in the structure. We would like our surrogate  $f(x)$  to accurately predict these stresses. An objective used in many machine learning problems is maximum likelihood estimation. In other words, we choose the parameters  $\theta$  (weights and biases in this case) to maximize the probability of observing the output data conditioned on our inputs  $x$ .

$$\max_{\theta} p(y|x; \theta) \quad (10.23)$$

This is not conditioned on the parameters  $\theta$  because we do not assume that those are also random variables. If we assume that the inputs are independently drawn, as is typical, then the probability can be expressed as a product across all the samples in the training data:

$$\max_{\theta} \prod_{i=1}^n p(y^{(i)}|x^{(i)}; \theta) \quad (10.24)$$

We now take the log of the objective, which does not change the solution, but changes the products to a better numerically behaved summation. We also add a negative sign up front so that the problem is one of minimization:

$$\min_{\theta} \sum_{i=1}^n -\log(p(y^{(i)}|x^{(i)}; \theta)) \quad (10.25)$$

This is a typical cost function, but if we further assume that the probability distribution is normally distributed (Gaussian), then this cost function simply reduces to a familiar sum of square errors:

$$\min_{\theta} \sum_{i=1}^n (f(x^{(i)}) - y^{(i)})^2 \quad (10.26)$$

We now have the objective and design variables in place to train the neural net. Like the other models discussed in this chapter it is critical to hold out some of the training data for cross validation. Two unique considerations for neural nets involve derivative computation, and the optimization algorithm used for training. We consider these topics next.

Because neural networks have many inputs but generally only have one output function, they are well suited to using reverse mode algorithmic differentiation (Section 6.6) to compute gradients. The reverse mode AD is so common that in the machine learning community it is simply called *back propagation*. However, for machine learning practitioners, back propagation is not usually performed at the code level as it is in general reverse mode AD, but rather is defined for larger sets of operations often requiring the user to use specialized libraries or allowing them define their own adjoints. While less general, this approach can enable increased efficiency and stability. The ReLU activation function (bottom of Fig. 10.16) is not differentiable right at  $z = 0$ , but in practice this is generally not problematic. Especially, because these methods typically rely on inexact gradients anyway as discussed next.

The form of the objective discussed in this section, and in many other machine learning problems is of the form:

$$f(x) = \sum_{i=1}^n \hat{f}(x^{(i)}) \quad (10.27)$$

where  $x^{(i)}$  is the  $i^{\text{th}}$  sample from the training set and  $\hat{f}$  is any function that operates on one training sample. As seen in this section, the objectives commonly used for many machine learning applications fit this form (e.g., negative log likelihood, or a squared error). The difficulty with these problems is that we often have large training sets, sometimes with  $n$  in the billions. That means that computing the objective can be time consuming, but computing the gradient is even more time consuming.

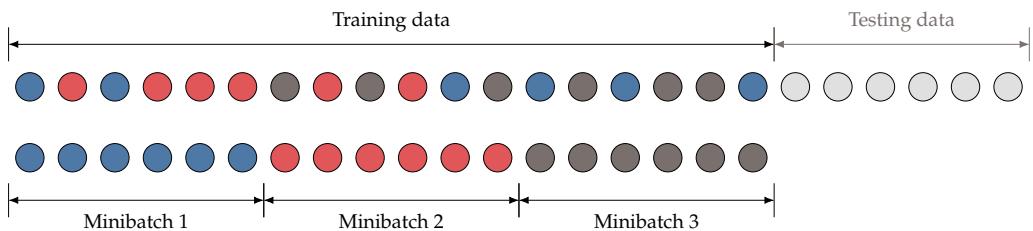
If we divide the objective by  $n$  (which does not change the solution), we can see that objective function is an expectation:

$$f(x) = \frac{1}{n} \sum_{i=1}^n \hat{f}(x^{(i)}) \quad (10.28)$$

We know that we can reasonably estimate an expected value from a smaller set of random samples. We call this subset of samples a *minibatch*  $\mathcal{S} = \{x^{(1)} \dots x^{(m)}\}$  where  $m$  is usually between one to a few hundred. The entries  $1 \dots m$  do not correspond to the first  $n$  entries but are drawn randomly from a uniform probability distribution (Fig. 10.17). Using the minibatch we can estimate the gradient as the sum of the gradients for each training point:

$$\nabla_x f(x) \approx \frac{1}{m} \sum_{i \in \mathcal{S}} \nabla_{x^{(i)}} \hat{f}(x^{(i)}) \quad (10.29)$$

Thus, we divide our training data into these minibatches and use a new minibatch to estimate the gradients at each iteration in the optimization.



This approach works well for these specific problems because of the unique form for the objective. As an example, if there were one

**Figure 10.17:** A simplified example of how training data is randomly assigned into minibatches.

million training samples then a single gradient evaluation would require evaluating all one million training samples. Alternatively, for a similar cost, a minibatch approach could update the design variables a million times using the gradient estimated from one training sample at a time. This latter process often converges much faster, especially because in these problems we generally do not care if we arrive at the absolute minimum.

Typically, this gradient is used with steepest descent (Section 4.4.1), also known as gradient descent. As discussed in Chapter 4, steepest descent is usually not very effective, however, for machine learning applications stochastic gradient descent has been found to work very well. This suitability is primarily because many machine learning optimizations are performed repeatedly, the true objective is often difficult to formalize, and finding the absolute minimum is not as important as finding a good enough solution quickly. One key difference in stochastic gradient descent is that we do not perform a line search. Rather the step size, called the *learning rate* in these applications, is a preselected size (often chosen somewhat arbitrarily) that is usually decreased between major iterations in the optimization.

Stochastic minibatching is easily applied to first-order methods and has thus driven innovation in alternative first-order methods that improve on stochastic gradient descent like Momentum, Adam, AMSGrad, etc.<sup>120</sup> While some of these methods may seem rather ad hoc there is mathematical rigor to many of them.<sup>121</sup> Batching makes the gradients noisy and so second-order methods are generally not pursued. However, ongoing research is exploring stochastic batch approaches that might effectively leverage the benefits of second-order methods.

## 10.6 Summary

Surrogate-based optimization can be a particularly effective way to incorporate simulations that are expensive or noisy. The first step in surrogate construction is sampling, in which we select evaluation points for constructing an initial surrogate. Full grid searches are too expensive for even a modest number of variables and so we need techniques that provide good coverage with a small number of samples. A popular technique for this application is LHS (which itself requires solving an optimization problem).

The next step is surrogate selection and construction. Linear regression models are popular approaches that cover a wide variety of models. Despite the name linear, these surrogates are generally highly nonlinear functions, but are linear in the coefficients (which is what

<sup>120</sup>. Ruder, *An overview of gradient descent optimization algorithms*. 2016

<sup>121</sup>. Goh, *Why Momentum Really Works*. 2017

we are selecting in the model construction process). Some common examples of linear regression models include polynomial models and Kriging (which is a subset of Gaussian radial basis functions). Data is used to train the model and select appropriate coefficients (an optimization problem). Cross validation is a critical component of this process. What we really want is good predictive capability, which means that the models work well on data that the model has not been trained against. Model selection often involves tradeoffs of more rigid models that do not need as much training data, versus more flexible models that require more training data.

The last step of the process is infill where points sampled during the process of optimization are used to update the surrogate. Some approaches are exploitation based, where we perform optimization using the surrogate, and use the optimal solution to update the model. For other models where uncertainty estimates are provided, exploration-based approaches can be used where we sample not just at the deterministic optimum, but at points where the expected improvement is high.

Finally, we discussed deep neural nets, which is another common surrogate model. While the general process is similar, there are some unique considerations. Neural nets are extremely flexible, but the downside of such flexibility is that large amounts of training data are needed to produce useful models. Approaches like backpropagation and stochastic gradients are needed to efficiently manage the large amount of training data.

## Problems

10.1 Answer *true* or *false* and justify your answer.

- a) You should use surrogate-based optimization when a problem has an expensive simulation and many design variables because it is immune to the “curse of dimensionality”.
- b) LHS is a random process that is more efficient than pure random sampling.
- c) LHS seeks to minimize the distance between the samples with the constraint that the projection on each axis follow a chosen probability distribution.
- d) Polynomial regressions are not considered to be surrogate models because they are too simple and do not consider any of the model physics.

- e) There can be some overlap between the training points and cross-validation points, as long as that overlap is small.
- f) Cross-validation is a required step in surrogate-based optimization.
- g) The more points you use to train a surrogate model, the more accurate it gets.
- h) In addition to modeling the function values, Kriging surrogate models also provide an estimate of the uncertainty in the values.
- i) A prediction-based exploitation infill strategy adds an infill point wherever the surrogate predicts the largest error.
- j) Maximizing the expected improvement maximizes the probability of finding a better function value.
- k) Neural networks require many nodes with a variety of sophisticated activation functions to represent challenging nonlinear models.
- l) Back propagation is the computation of the derivatives of the neural net output with respect to the activation function weights using reverse mode AD.

10.2 *Latin hypercube sampling*. Use a LHS package to create and plot 20 points across two dimensions with uniform projection in both dimensions.

10.3 *Inversion sampling*. Use inversion sampling with Latin hypercube sampling to create and plot 100 points across two dimensions. Each dimension should follow a normal distribution with zero mean and a standard deviation of 1 (cross-terms in covariance matrix are 0).

10.4 *Linear regression*. Use the following training data sampled at  $x$  with resulting function value  $f$ :

$$\begin{aligned} x = [ & -2.0000, -1.7895, -1.5789, -1.3684, -1.1579, \\ & -0.9474, -0.7368, -0.5263, -0.3158, -0.1053, \\ & 0.1053, 0.3158, 0.5263, 0.7368, 0.9474, \\ & 1.1579, 1.3684, 1.5789, 1.7895, 2.0000 ] \end{aligned}$$

$$\begin{aligned} f = [ & 7.7859, 5.9142, 5.3145, 5.4135, 1.9367, \\ & 2.1692, 0.9295, 1.8957, -0.4215, 0.8553, \\ & 1.7963, 3.0314, 4.4279, 4.1884, 4.0957, \\ & 6.5956, 8.2930, 13.9876, 13.5700, 17.7481 ] \end{aligned}$$

Use linear regression to determine the coefficients for a polynomial basis of  $[x^2, x, 1]$  to predict  $f(x)$ . Plot your fit against the training data and report the coefficients for the polynomial bases.

- 10.5 *Cross validation.* Use the following training data sampled at  $x$  with resulting function value  $f$ :

$$\begin{aligned} x = [ & -3.0, -2.6053, -2.2105, -1.8158, -1.4211, \\ & -1.0263, -0.6316, -0.2368, 0.1579, 0.5526, \\ & 0.9474, 1.3421, 1.7368, 2.1316, 2.5263, \\ & 2.9211, 3.3158, 3.7105, 4.1053, 4.5] \end{aligned}$$

$$\begin{aligned} f = [ & 43.1611, 28.1231, 12.9397, 3.7628, -2.5457, \\ & -4.267, 2.8101, -0.6364, 1.1996, -0.9666, \\ & -2.7332, -6.7556, -9.4515, -7.0741, -7.6989, \\ & -8.4743, -7.9017, -2.0284, 11.9544, 33.7997] \end{aligned}$$

- a) Create a polynomial surrogate model using the set of polynomial basis functions  $x^i$  for  $i = 0 : n$ . Plot the error in the surrogate model while increasing  $n$  (the maximum order of the polynomial model) from 1 to 20.
- b) Plot the polynomial fit for  $n = 16$  against the data and comment on its suitability.
- c) Recreate the error plot versus polynomial order using k-fold cross validation with ten divisions. Be sure to limit the y-axes to the area of interest.
- d) Plot the polynomial fit against the data for a polynomial order that produces low error under cross validation, and report the coefficients for the polynomial. Justify your selection.

- 10.6 *Wave drag minimization using a surrogate.* Minimize the drag of a supersonic body of revolution using a global polynomial surrogate model. The provided analysis code is somewhat noisy, similar to what might exist with experimental data or with some grid-based simulations, hence the use of a surrogate. The details of this problem are [here](#).

Present your methodology and discuss your results and lessons learned.

General nonlinear optimization problems are difficult to solve. Depending on the particular optimization algorithm, they require may the selection of tuning parameters, derivatives, appropriate scaling, and trying different starting points. Convex optimization problems do not have any of those issues and are thus relatively easy to solve. The difficulty is that some strict requirements must be met. Even for candidate problems that have the potential to be convex, significant experience is often needed to recognize and utilize techniques that reformulate the problems into an appropriate form.

By the end of this chapter you should be able to:

1. Understand the benefits and limitations of convex optimization.
2. Identify and solve linear and quadratic optimization problems.
3. Formulate and solve convex optimization problems.
4. Identify and solve geometric programming problems.

## 11.1 Introduction

Convex optimization problems have desirable characteristics that make them more predictable and easier to solve. Since a convex problem has provably only one optimum, convex optimization methods always converge to the global minimum. Solving convex problems is straightforward and does not require a starting point, parameter tuning, or derivatives, and they can scale efficiently even for problems with millions of design variables.<sup>122</sup> All we need to solve a convex problem is set it up properly; there is no need to worry about convergence, local optimum, or noisy functions. Some of the convex problems are so straightforward to solve that they are often not recognized as an

<sup>122</sup>. Diamond *et al.*, *Convex Optimization with Abstract Linear Operators*. 2015

optimization problem and are just thought of as a function or operation. A familiar example of convex optimization is the linear-least-squares problem (described in a subsequent section).

While these are very desirable properties, the catch is that for an optimization problem to be convex, it must satisfy some strict requirements. Namely, the objective and all inequality constraints must be convex functions, and the equality constraints must be affine.\* A function  $f$  is convex if:

$$f((1-\eta)x_1 + \eta x_2) \leq (1-\eta)f(x_1) + \eta f(x_2) \quad (11.1)$$

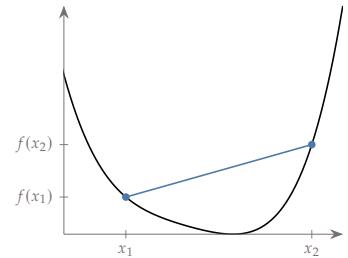
for all  $x_1$  and  $x_2$  in the domain, where  $0 \leq \eta \leq 1$ . This requirement is illustrated in Fig. 11.1 for the one-dimensional case. The right-hand side of the inequality is just the equation of a line from  $f(x_1)$  to  $f(x_2)$  (the blue line), whereas the left-hand side is the function  $f(x)$  evaluated at all points between  $x_1$  to  $x_2$  (the black curve). The inequality says that the function must always be below a line joining any two points in the domain. Stated informally, a convex function looks something like a bowl everywhere.

Unfortunately, even these strict requirements are not enough. In general, we cannot identify a given problem as convex or take advantage of its structure to solve it efficiently, and thus must treat it as a general nonlinear problem. There are two approaches to take advantage of convexity. The first one is to directly formulate the problem in a known convex form, such as a linear program or a quadratic program (discussed later in this chapter). The second option is to use *disciplined convex programming*, which is a specific set of rules and mathematical functions that one can use to build up a convex problem. By following these rules, one can always translate the problem into an efficiently solvable form automatically.

While both of these approaches are straightforward to apply, they also expose the main weakness of these methods: we need to be able to express the objective and inequality constraints using only these elementary functions and operations. In most cases, this requirement means that the model must be simplified. Often, a problem is not directly expressed in a convex form and a combination of experience and creativity is needed to reformulate the problem in an equivalent manner that is convex.

Simplifying models usually results in a reduction in fidelity. This is less problematic for optimization problems that are intended to be solved repeatedly, such as in optimal control and machine learning, domains in which convex optimization is heavily used. In these cases, simplification by local linearization, for example, is less problematic

\*An affine function consists of a linear transformation and a translation. Informally, this type of function is often referred to as linear (including in this book), but strictly speaking these are distinct concepts. For example:  $Ax$  is a linear function in  $x$ , whereas  $Ax + b$  is an affine function in  $x$ .



**Figure 11.1:** Illustration of what it means for a function to be convex in the one-dimensional case. The function (black) must be below a line that connects any two points (blue) in the domain.

because the linearization can be updated in the next time step. However, this reduction in fidelity is problematic for design applications. In design scenarios, the optimization is performed once, and the design cannot continue to be updated after it is created. For this reason, convex optimization less frequently used for design applications, with the exception of some limited uses of geometric programming, a topic discussed in more detail in Section 11.6.

This chapter is introductory in nature, focusing only on understanding what convex optimization is useful for and describing some of the most widely used forms.<sup>†</sup> The known categories of convex optimization problems include: linear programming, quadratic programming, second-order cone programming, semidefinite programming, cone programming, and graph form programming. Each of these categories is a subset of the next (Fig. 11.2). We will focus on the first three because they are the mostly widely used, including in other chapters in this book.<sup>‡</sup> The latter three forms are less frequently formulated directly. Instead, the user applies elementary functions and operations, rules specified by disciplined convex programming, and a software tool transforms the problem into a suitable conic form that can be solved. This procedure is described in Section 11.5.

After covering those main categories of convex optimization we discuss geometric programming. Geometric programming problems are actually not convex, but with a change of variables, they can be transformed into an equivalent convex form extending the types of problems that can be solved with convex optimization.

## 11.2 Linear Programming

A *linear program* (LP) is an optimization problem with linear objective and linear constraints and can be written as

$$\begin{aligned} & \text{minimize} && f^T x \\ & \text{subject to} && Ax + b = 0 \\ & && Cx + d \leq 0, \end{aligned} \tag{11.2}$$

where,  $f$ ,  $b$ , and  $d$  are vectors and  $A$  and  $C$  are matrices. All LPs are convex.

**Example 11.1:** Formulating a linear programming problem.

Suppose we are going shopping and want to figure out how to best meet our nutritional needs for the least amount of cost. We enumerate all the food options, and use the variable  $x_j$  to represent how much of food  $j$  we will purchase. The parameter  $c_j$  is the cost of a unit amount of food  $j$ . The

<sup>†</sup>Boyd *et al.*<sup>123</sup> provides a good starting point for those seeking a more complete introduction into the field of convex optimization.

<sup>123</sup> Boyd *et al.*, *Convex Optimization*. 2004

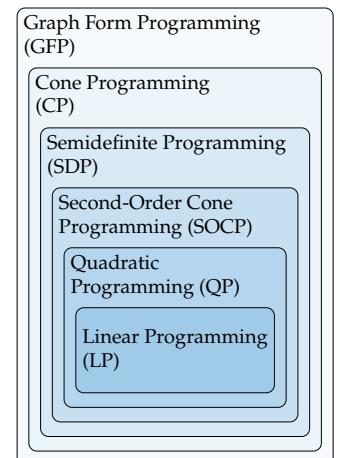
<sup>‡</sup>Many good references exist with examples for those categories we do not discuss in detail.<sup>124–125</sup>

<sup>124</sup> Lobo *et al.*, *Applications of second-order cone programming*. 1998

<sup>125</sup> Vandenberghe *et al.*, *Applications of semidefinite programming*. 1999

<sup>126</sup> Vandenberghe *et al.*, *Semidefinite Programming*. 1996

<sup>127</sup> Parikh *et al.*, *Block splitting for distributed optimization*. 2013



**Figure 11.2:** Relationship between various convex optimization problems.

parameter  $N_{ij}$  is the amount of nutrient  $i$  contained in a unit amount of food  $j$ . We need to make sure we have at least  $r_i$  of nutrient  $i$  to meet our dietary requirements. We can now formulate this as an optimization problem. We wish to minimize the cost of our food:

$$\text{minimize} \quad \sum_j c_j x_j = c^T x \quad (11.3)$$

To meet the nutritional requirement of nutrient  $j$  we need to satisfy:

$$\sum_j N_{ij} x_j \geq r_i \Rightarrow Nx \geq r. \quad (11.4)$$

Finally, we cannot purchase a negative amount of food so we require  $x \geq 0$ . The objective and all of the constraints are linear in  $x$ , so this is an LP ( $f = c$ ,  $C = -N$ ,  $d = r$ ). We do not need to artificially restrict what foods we include in our initial list of possibilities. The formulation allows the optimizer to select a given food item  $x_i$  to be zero (that is, do not purchase any of that food item), according to what is optimal.

As a concrete example, let us consider a simplified version (and a reductionist view of nutrition) with 10 food options and three nutrients with the amounts shown below.

Food	Cost	Nutrient 1	Nutrient 2	Nutrient 3
A	0.46	0.56	0.29	0.48
B	0.54	0.84	0.98	0.55
C	0.40	0.23	0.36	0.78
D	0.39	0.48	0.14	0.59
E	0.49	0.05	0.26	0.79
F	0.03	0.69	0.41	0.84
G	0.66	0.87	0.87	0.01
H	0.26	0.85	0.97	0.77
I	0.05	0.88	0.13	0.13
J	0.60	0.62	0.69	0.10

If we call the amount of each food  $x$ , the cost column  $c$ , and the nutrient columns  $n_1, n_2, n_3$  then we can setup the following linear problem:

$$\begin{aligned} & \text{minimize} \quad c^T x \\ & \text{subject to} \quad 5 \leq n_1^T x \leq 8 \\ & \quad 7 \leq n_2^T x \\ & \quad 1 \leq n_3^T x \leq 10 \\ & \quad x \leq 4 \end{aligned} \quad (11.5)$$

The last constraint was added to ensure we do not eat too much of any one item and get tired of it. LP solvers are widely available. In fact, some solvers

operate independent of a programming language as the input is just a table of numbers. The solution in this case is:

$$x = [0, 1.43, 0, 0, 0, 4.00, 0, 4.00, 0.73, 0]^T \quad (11.6)$$

suggesting that our optimal diet consists of items B, F, H, and I in the proportions shown above. The solution hit the upper limit on nutrient 1 and the lower limit on nutrient 2.

---

LPs frequently occur with allocation or assignment problems, such as choosing an optimal portfolio of stocks, deciding what mix of products to build, deciding what tasks should be assigned to each worker, determining which goods to ship to which locations. These types of problems occur frequently in domains like operations research, finance, supply chain management, and transportation.

A common consideration with LPs is whether or not the variables should be discrete. In Ex. 11.1,  $x_i$  is a continuous variable and purchasing fractional amounts of food may or may not make sense, depending on the type of food. If we were performing an optimal stock allocation then we can purchase fractional amounts of stock, but if we were optimizing how much of each product to manufacture, it might not make sense to build 32.4 products. In these cases, we may want to restrict the variables to be integers, which are called integer constraints. These types of problems require discrete optimization algorithms, which are covered in Chapter 8.

### 11.3 Quadratic Programming:

A *quadratic program* (QP) has a quadratic objective and linear constraints. Quadratic programming was mentioned in Section 5.4 when discussing sequential quadratic programming. A general QP can be expressed as:

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Qx + f^T x \\ & \text{subject to} && Ax + b = 0 \\ & && Cx + d \leq 0 \end{aligned} \quad (11.7)$$

A QP is only convex if the matrix  $Q$  is positive semidefinite. A QP is reduced to an LP if  $Q = 0$ .

One of the most common QP examples (really a subset of QP) is least-squares regression, which is used in many applications, such as data fitting. As the name suggests, least squares seeks to minimize the

sum of squared residuals:

$$\text{minimize} \quad \sum_i (\hat{b}_i - b_i)^2, \quad (11.8)$$

where the vector  $\hat{b}$  contains the estimated values (from a model, for example), and  $b$  contains the data points that we are trying to fit. If we assume a linear model, then  $\hat{b} = Ax$ , where  $x$  are the model parameters we want to optimize to fit the data. Here, “linear” means linear in the coefficients, not in the data fit. For example, we could estimate the coefficients  $c_i$  of a quadratic function that best fits some data:

$$f(\zeta) = c_1\zeta^2 + c_2\zeta + c_3 \quad (11.9)$$

This equation is linear in the coefficients (which corresponds to  $x = [c_1, c_2, c_3]$ ). For this to be a least-squares problem,  $A$  must have more rows than columns, that is, more equations than unknowns, and is also known as overdetermined.

We can rewrite the problem statement as:

$$\text{minimize} \quad \sum_i (a_i^T x - b_i)^2, \quad (11.10)$$

where  $a_i^T$  is the  $i^{\text{th}}$  row in the matrix  $A$ . Equivalently, we can express the least-squares problem as minimizing the square of a 2-norm:

$$\text{minimize} \quad \|Ax - b\|_2^2 \quad (11.11)$$

which can be expressed equivalently as

$$\|Ax - b\|_2^2 = (Ax - b)^T (Ax - b) = x^T A^T A x - 2b^T A x + b^T b \quad (11.12)$$

This is the same as the general QP form, where  $Q = 2A^T A$ ,  $f = -2A^T b$ , and  $b^T b$  is just a constant and so does not affect the optimal solution. Thus, least squares is an unconstrained QP. Least squares actually has an analytic solution if  $A$  has full rank ( $x^* = (A^T A)^{-1} A^T b$ ), so the machinery of a QP is not necessary. However, we can add constraints in QP form to solve *constrained least squares* problems, which generally do not have analytic solutions.

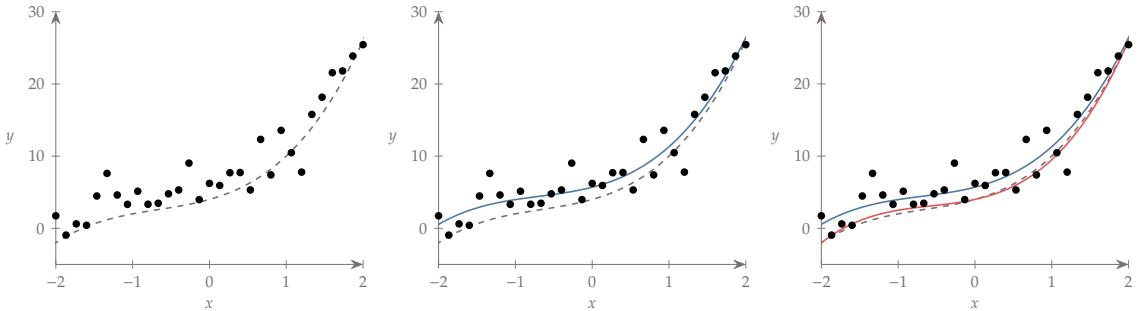
---

**Example 11.2:** A constrained least-squares QP.

The left pane of Fig. 11.3 shows some example data that is both noisy and biased relative to the true (but unknown) underlying curve represented as a dashed line. Given the data points we would like to estimate the underlying functional relationship. We assume that the relationship is cubic:

$$y(x) = a_1 x^3 + a_2 x^2 + a_3 x + a_4 \quad (11.13)$$

and need to estimate the coefficients  $a_1, \dots, a_4$ . As discussed above this can be posed as a QP problem, or even more simply as an analytic problem. The resulting least-squares fit is shown in middle pane of Fig. 11.3.



Suppose from some careful measurements, or additional data, we know an upper bound on the function value at a few places. For this example we assume that we know that  $f(-2) \leq -2$ ,  $f(0) \leq 4$ ,  $f(2) \leq 26$ . These requirements can be posed as linear constraints:

$$\begin{bmatrix} (-2)^3 & (-2)^2 & -2 & 1 \\ 0 & 0 & 0 & 1 \\ 2^3 & 2^2 & 2 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \leq \begin{bmatrix} -2 \\ 4 \\ 26 \end{bmatrix} \quad (11.14)$$

We add these linear constraints to our quadratic objective (minimizing the sum of the squared error) and the resulting problem is still a QP. The resulting solution is shown in the right pane of Fig. 11.3, which results in a much more accurate fit.

**Figure 11.3: hello**

---

**Example 11.3: Linear-quadratic regulator (LQR) controller.**

Another common example of a QP occurs in optimal control. Consider a discrete-time linear dynamic system:

$$x_{t+1} = Ax_t + Bu_t \quad (11.15)$$

where  $x_t$  is the deviation from a desired state at time  $t$  (for example, the positions and velocities of an aircraft), and  $u_t$  are the control inputs that we want to optimize (for example, control surface deflections). The above dynamic equation can be used as a set of linear constraints in an optimization, but we must decide on an objective.

One would like to have small  $x_t$  because that would mean reducing the error in our desired state quickly, but we would also like to have small  $u_t$  because small control inputs require less energy. These are competing objectives, where a small control input will take longer to minimize error in a state, and vice-versa.

One way to express this objective is as a quadratic function:

$$\text{minimize} \quad \frac{1}{2} \sum_{t=0}^N (x_t^T Q x_t + u_t^T R u_t), \quad (11.16)$$

where the weights in  $Q$  and  $R$  reflect our preferences on how important it is to have small state error versus small control inputs. (This is an example of a multiobjective function, which we explained in Chapter 9) The equation has a form like kinetic energy, and the LQR problem could be thought of as determining the control inputs that minimize the energy expended, subject to the vehicle dynamics. This particular choice of objective was intentional because it means that the problem is a convex QP (as long as we choose positive weights). Because it is convex, this problem can be solved reliably and efficiently, both necessary conditions for a robust control law.

---

## 11.4 Second-Order Cone Programming:

A *second-order cone program* (SOCP) has a linear objective and a second-order cone constraint.

$$\begin{aligned} & \text{minimize} && f^T x \\ & \text{subject to} && \|A_i x + b_i\|_2 \leq c_i^T x + d_i \\ & && Gx + h = 0 \end{aligned} \quad (11.17)$$

If  $A_i = 0$  then this form reduces to a linear programming problem.

One useful subset of SOCP is a *quadratically-constrained quadratic program* (QCQP). A QCQP is the same as a QP but with the addition of quadratic inequality constraints instead of linear ones, that is,

$$\begin{aligned} & \text{minimize} && \frac{1}{2} x^T Q x + f^T x \\ & \text{subject to} && Ax + b = 0 \\ & && \frac{1}{2} x^T R_i x + c_i^T x + d_i \leq 0 \text{ for } i = 1, \dots, m \end{aligned} \quad (11.18)$$

Both  $Q$  and  $R$  must be positive semidefinite for the QCQP to be convex. A QCQP reduces to a QP if  $R = 0$ . We saw QCQPs when solving trust-region problems Section 4.5, although for trust-region problems only an approximate solution method is typically used.

Every QCQP can be expressed as a SOCP (though not vice-versa). The QCQP in Eq. 11.18 can be written in this equivalent form:

$$\begin{aligned} & \text{minimize} && y \\ & \text{subject to} && \|Fx + g\|_2 \leq y \\ & && Ax + b = 0 \\ & && \|G_i x + h_i\|_2 \leq 0 \end{aligned} \tag{11.19}$$

If we square both sides of the first and last constraint, we see that this formulation is exactly equivalent to the QCQP where  $Q = 2F^T F$ ,  $f = 2F^T g$ ,  $R_i = 2G_i^T G_i$ ,  $c_i = 2G_i^T h_i$  and  $d_i = h_i^T h_i$ . The matrices  $F$  and  $G_i$  are the square roots of the matrices  $Q$  and  $R_i$  respectively (divided by two), and would be computed from a factorization.

## 11.5 Disciplined Convex Optimization

Disciplined convex optimization allows us to build convex problems using a specific set of rules and mathematical functions. By following this set of rules, the problem can be translated automatically in a form that can be efficiently solved using convex optimization algorithms.<sup>§</sup>

The following are examples of functions that are convex:

- Exponential functions:  $e^{ax}$  where  $a$  is any real number
- Power functions:  $x^a$  for  $a \geq 1$  or  $a \leq 0$  or  $-x^a$  for  $0 \leq a \leq 1$
- Negative logarithms:  $-\log(x)$
- Norms, including absolute value:  $\|x\|$
- Maximum function:  $\max(x_1, x_2, \dots, x_n)$
- log-sum-exp:  $\log(e^{x_1} + e^{x_2} + \dots + e^{x_n})$

The functions do not need to be continuously differentiable because this is not a requirement of convexity.

A disciplined convex problem can be formulated using any of these functions for our objective or inequality constraints. We can also use various operations that preserve convexity to build up more complex expressions. Some of the more common operations are:

- Multiplying a convex function by a positive constant
- Adding convex functions
- Composing a convex function with an affine function, i.e., if  $f(x)$  is convex, then  $f(Ax + b)$  is also convex

<sup>§</sup>Grant *et al.*<sup>128</sup> shows how the disciplined convex problem rules allow for translating the problem into an efficiently solvable form automatically.

<sup>128</sup>. Grant *et al.*, *Disciplined Convex Programming*. 2006

- Taking the maximum of two convex functions

While these functions and operations greatly expand the types of convex problems that we can solve beyond LPs and QPs, they are still restrictive within the broader scope of nonlinear programming. Still, for objectives and constraints that require only simple mathematical expressions, there is a possibility that it can be posed as a disciplined convex optimization problem. The original expression of a problem is often not convex, but can be made convex through a transformation to a mathematically equivalent problem. Some of these transformation techniques, including performing a change of variables, adding slack variables, or expressing the objective in a different form. Successfully recognizing and applying these techniques is a skill that takes practice.

---

**Tip 11.4:** Software for disciplined convex programming.

CVX and its variants are free popular tools for disciplined convex programming with interfaces for multiple programming languages.<sup>¶</sup>

<sup>¶</sup><https://stanford.edu/~boyd/software.html>

---

## 11.6 Geometric Programming

A *geometric program* (GP) is not convex, but can be transformed into an equivalent convex problem. GPs are defined using monomials and posynomials. A monomial is a function of the form:

$$f(x) = c x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \quad (11.20)$$

where  $c > 0$  and all  $x_i > 0$ . A posynomial is a sum of monomials:

$$f(x) = \sum_{j=1}^N c_j x_1^{a_{1j}} x_2^{a_{2j}} \cdots x_n^{a_{nj}} \quad (11.21)$$

where all  $c_j > 0$ .

---

**Example 11.5:** Monomials and posynomials in engineering.

Monomials and posynomials appear in many engineering expressions. For example, the calculation of lift from the definition of the lift coefficient is a monomial:

$$L = C_L \frac{1}{2} \rho V^2 S \quad (11.22)$$

Total incompressible drag, a sum of parasitic and induced drag, is a posynomial:

$$D = C_D p q S + \frac{C_L^2}{\pi A Re} q S \quad (11.23)$$

---

A GP in standard form is written as:

$$\begin{aligned} & \text{minimize} && f_0(x) \\ & \text{subject to} && f_i(x) \leq 1 \\ & && h_i(x) = 1 \end{aligned} \tag{11.24}$$

where all of the  $f_i$  are posynomials and the  $h_i$  are monomials. This problem does not fit into any of the convex optimization problems defined in the previous section, and it is not convex. The reason why this formulation is useful is that we can convert it into an equivalent convex optimization problem.

First, we take the logarithm of the objective and of both sides of the constraints:

$$\begin{aligned} & \text{minimize} && \log f_0(x) \\ & \text{subject to} && \log f_i(x) \leq 0 \\ & && \log h_i(x) = 0. \end{aligned} \tag{11.25}$$

Let us further examine the equality constraints. Recall that  $h_i$  is a monomial, so writing one of the constraints explicitly results in the form:

$$\log(cx_1^{a_1}x_2^{a_2}\dots x_n^{a_n}) = 0 \tag{11.26}$$

Using the properties of logarithms, this can be expanded to an equivalent expression:

$$\log c + a_1 \log x_1 + a_2 \log x_2 + \dots + a_n \log x_n = 0 \tag{11.27}$$

Introducing a change of variables,  $y_i = \log x_i$ , results in the following equality constraint:

$$\begin{aligned} & a_1 y_1 + a_2 y_2 + \dots + a_n y_n + \log c = 0 \\ & a^T y + \log c = 0 \end{aligned} \tag{11.28}$$

This is an affine constraint in  $y$ .

The objective and inequality constraints are more complex because they are posynomials. The expression  $\log f_i$  written in terms of a posynomial results in:

$$\log \left( \sum_{j=1}^N c_j x_1^{a_{1j}} x_2^{a_{2j}} \dots x_n^{a_{nj}} \right) \tag{11.29}$$

Because this is a sum of products, we cannot use the logarithm to expand each term. However, we still introduce the same change of variables (expressed as  $x_i = e^{y_i}$ ):

$$\begin{aligned}\log f_i &= \log \left( \sum_{j=1}^N c_j e^{y_1 a_{1j}} e^{y_2 a_{2j}} \dots e^{y_n a_{nj}} \right) \\ &= \log \left( \sum_{j=1}^N c_j e^{y_1 a_{1j} + y_2 a_{2j} + \dots + y_n a_{nj}} \right) \\ &= \log \left( \sum_{j=1}^N e^{a_j^T y + b_j} \right) \text{ where } b_j = \log c_j.\end{aligned}\tag{11.30}$$

This is a log-sum-exp of an affine function. As mentioned in the previous section, log-sum-exp is convex, and a convex function composed with an affine function is a convex function. Thus, the objective and inequality constraints are convex in  $y$ . Because the equality constraints are affine, we have a convex optimization problem obtained through a change of variables.

Geometric programming has been successfully used for aircraft design applications using relationships like the simple ones shown in Ex. 11.5.<sup>129</sup>

Unfortunately, many other functions do not fit this form (e.g., design variables that can be positive or negative, terms with negative coefficients, trigonometric functions, logarithms, exponents). GP modelers use various techniques to extend usability including using a Taylor's series across a restricted domain, fitting functions to posynomials,<sup>130</sup> and rearranging expressions to other equivalent forms including implicit relationships. A good deal of creativity and some sacrifice in fidelity is usually needed to create a corresponding GP from a general nonlinear programming problem. Still, if the sacrifice in fidelity is not too great, there is a big upside as it comes with all the benefits of convexity (guaranteed convergence, global optimality, efficiency, no parameter tuning, and limited scaling issues).

One extension to geometric programming is signomial programming. A signomial program has the same form except that the coefficients  $c_i$  can be positive or negative (the design variables  $x_i$  must still be strictly positive). Unfortunately, this problem cannot be transformed to a convex one, so it can no longer guarantee a global optimum. Still, a signomial program can usually be solved using a sequence of geometric programs, so it is much more efficient than solving the general nonlinear problem. Signomial programs have been used to extend the range

<sup>129</sup> Hoburg et al., *Geometric Programming for Aircraft Design Optimization*. 2014

<sup>130</sup> Hoburg et al., *Data fitting with geometric-programming-compatible softmax functions*. 2016

of design problems that can be solved using geometric programming techniques.<sup>131,132</sup>

#### Tip 11.6: Software for Geometric Programming

GPKit<sup>¶</sup> is a useful, freely available software package for posing and solving geometric programming (and signomial programming) models.

131. Kirschen *et al.*, *Application of Signomial Programming to Aircraft Design*. 2018

132. York *et al.*, *Turbofan Engine Sizing and Tradeoff Analysis via Signomial Programming*. 2018

<sup>¶</sup><https://gpkit.readthedocs.io>

## 11.7 Summary

Convex optimization problems are highly desirable as they do not require any parameter tuning, starting points, derivatives, and converge reliably and rapidly to the global optimum. The tradeoff is that the form of the objective and constraints must meet stringent requirements. These requirements often necessitate simplifying the physics models and often require clever reformulations. The reduction in model fidelity is still well suited to domains where optimizations are performed repeatedly in time (e.g., controls, machine learning), or for high-level conceptual design studies. Linear programming and quadratic programming in particular are widely used across many domains and form the bases of many of the gradient-based algorithms used to solve general non-convex problems.

## Problems

### 11.1 Answer *true* or *false* and justify your answer.

- a) The optimum found through convex optimization is guaranteed to be the global optimum.
- b) Cone programming problems are a special case of quadratic programming problems.
- c) It is sometimes possible to obtain distinct feasible regions in linear optimization.
- d) A quadratic problem is a problem with quadratic objective and quadratic constraints.
- e) A quadratic problem is only convex if the Hessian of the objective function is positive definite.
- f) Solving a quadratic problem is easy because the solution can be obtained analytically.

- g) Least-square regression is a type of quadratic programming problem.
- h) Second-order cone programming problems feature a linear objective and a second-order cone constraint.
- i) Disciplined convex optimization builds convex problems by using convex differentiable functions.
- j) It is possible to transform some nonconvex problems into convex ones by using a change of variables, adding slack variables, or reformulating the objective function.
- k) A geometric program is not convex but can be transformed into an equivalent convex program.
- l) Convex optimization algorithms work well as long as a good starting point is provided.

11.2 Solve using a convex solver (not a general nonlinear solver).

$$\begin{aligned}
 & \text{minimize} && x_1^2 + 3x_2^2 \\
 & \text{subject to} && x_1 + 4x_2 \geq 2 \\
 & && 3x_1 + 2x_2 \geq 5 \\
 & && x_1 \geq 0, x_2 \geq 0
 \end{aligned}$$

11.3 The following foods are available to you at your nearest grocer.

Food	Cost	Nutrient 1	Nutrient 2	Nutrient 3
A	7.68	0.16	1.41	2.40
B	9.41	0.47	0.58	3.95
C	6.74	0.87	0.56	1.78
D	3.95	0.62	1.59	4.50
E	3.13	0.29	0.42	2.65
F	6.63	0.46	1.84	0.16
G	5.86	0.28	1.23	4.50
H	0.52	0.25	1.61	4.70
I	2.69	0.28	1.11	3.11
J	1.09	0.26	1.88	1.74

Minimize the amount you spend while making sure you get at least 5 units of Nutrient 1, between 8 and 20 units of nutrient 2, and between 5 and 30 units of nutrient 3. Also be sure not to buy more than 4 units of any one food item, just for variety. Determine the optimal amount of each item to purchase and the total cost.

11.4 Consider the following simplified aircraft wing design problem. Our goal is to primarily size the wing area ( $S$ ), aspect ratio ( $AR$ ), and flight speed ( $V$ ), in order to minimize drag:

$$D = C_D \frac{1}{2} \rho V^2 S \quad (11.31)$$

where the drag coefficient is a sum of parasitic drag, lift-dependent drag, and drag of the rest of the aircraft.

$$C_D = k C_f \frac{S_{\text{wet}}}{S} + \frac{CL^2}{\pi AR e} + (CDS)_{\text{other}} \frac{1}{S} \quad (11.32)$$

The skin friction coefficient is a function of Reynolds number:

$$C_f = \frac{0.074}{Re^{0.2}} \quad (11.33)$$

where the Reynolds number is:

$$Re = \frac{\rho V \sqrt{S/AR}}{\mu} \quad (11.34)$$

We need to add some constraints. One is that lift equals weight:

$$W = C_L \frac{1}{2} \rho V^2 S \quad (11.35)$$

where the weight is a sum of the wing weight and the fixed weight of the rest of the aircraft:

$$W = W_w + W_{\text{other}} \quad (11.36)$$

and the wing weight is estimated using a statistical fit:

$$W_w = 45.42S + 8.71 \times 10^{-5} \frac{N_{\text{load}}(S \cdot AR)^{3/2} \sqrt{W_{\text{other}} W}}{S \tau} \quad (11.37)$$

Another constraint is that we need enough wing area to fly at our desired stall speed:

$$\frac{2W}{\rho V_s^2 S} \leq C_{L\max} \quad (11.38)$$

Variables that were not defined above are fixed parameters shown in the following table.

Parameter	Value	Unit	Description
$\rho$	1.23	$\text{kg}/\text{m}^3$	density of air
$\mu$	$1.78 \times 10^{-5}$	$\text{kg}/(\text{m sec})$	viscosity of air
$k$	1.2		form factor
$S_{\text{wet}}/S$	2.05		ratio of wetted area to reference area
$e$	0.96		Oswald efficiency factor
$(CDS)_{\text{other}}$	0.0306		drag area for rest of aircraft
$W_{\text{other}}$	4940	N	weight of rest of aircraft
$N_{\text{load}}$	2.5		load factor
$\tau$	0.12		airfoil thickness-to-chord ratio
$V_s$	22	m/s	stall speed
$C_{L_{\max}}$	2		maximum lift coefficient at landing

Formulate the above problem as a geometric program and solve it.<sup>\*\*</sup>

<sup>\*\*</sup>GPkit, <https://gpkit.readthedocs.io> is one good option to solve this. If you get stuck this problem is an example in the paper [Geometric Programming for Aircraft Design Optimization](#) as well as an example in the documentation for GPkit (with slightly different values).

Uncertainty is always present in engineering design. For example, manufacturing processes create deviations from the specifications, operating conditions vary from the ideal, and some parameters are inherently variable. Optimization with deterministic inputs can lead to poorly performing designs. To create robust and reliable designs, we must treat the relevant parameters and design variables as random variables. *Optimization under uncertainty* (OUU) is the optimization of systems in the presence of uncertain parameters or design variables.

By the end of this chapter you should be able to:

1. Define robustness and reliability in the context of optimization under uncertainty.
2. Describe and use several strategies for both robust optimization and reliability.
3. Understand the pros and cons for the following forward propagation methods: direct quadrature, Monte Carlo methods, first-order perturbation methods, and polynomial chaos.
4. Use some of the forward propagation methods within an optimization.

## 12.1 Introduction

We call a design *robust* if its performance is less sensitive to inherent variability. In other words, the *objective* function is less sensitive to variations in the random design variables and parameters. Similarly, we call a design *reliable* if it is less prone to failure under variability. In other words, the *constraints* have a lower probability of being violated under variations in the random design variables and parameters.\*

\*While we maintain a distinction in this book, much of the literature include both of these concepts under the umbrella of robust optimization.

---

**Example 12.1:** Robust versus reliable designs.

A familiar example of robust design occurs when playing the board game Monopoly. On a given turn, if you knew for certain where an opponent was going to land next, it would make sense to put all of your funds into developing that one property to its fullest extent. However, because their next position is uncertain, a better strategy might be to develop multiple nearby properties (each to a lesser extent because of a fixed monetary resource). This is an example of robust design: the expected return is less sensitive to input variability. However, because you develop multiple properties, a given property will have a lower return than if you had only developed one property. This is a fundamental tradeoff in robust design. An improvement in robustness generally is not free; instead, it requires a tradeoff in peak performance. This is known as a risk-reward tradeoff.

A familiar example of reliable design is when planning a trip to the airport. Experience suggests that it is not a good idea to use average times to plan your arrival down to the minute. Instead, if you want a high probability of making your flight, you plan for variability in traffic and security lines and add a buffer to your departure time. This is an example of a reliable design: it is less prone to failure under variability. Reliability is also not free, and generally requires a tradeoff in the objective (in this example, optimal use of time perhaps).

---

In this chapter, we first provide a brief review of some elements of statistics and probability theory. Next, we discuss how uncertainty can be used in the objective function allowing for robust designs, and how it can be used in constraints allowing for reliable designs. Finally, we discuss a few different methods for propagating input uncertainties through a computational model to produce output statistics, a process called forward propagation. This chapter only presents a introduction to these topics. Uncertainty quantification, and OUU, is a large and actively growing field.

## 12.2 Statistics Review

Imagine measuring the axial strength of a rod by perform a tensile test with many rods, each designed to be identical. Even with “identical” rods, every time you perform the test you get a different result (hopefully with relatively small differences). This variation has many potential sources including variation in the manufactured size and shape, in the composition of the material, in the contact between the rod and testing fixture. In this example, we would call the axial strength a *random variable*, and the result from one test would be a random sample. The random variable, axial strength, is a function of several

other random variables such as bar length, bar diameter, and material Young's modulus.

One measurement does not tell us anything about how variable the axial strength is, but if we perform the test many times we can learn a lot about its distribution. From this information we can infer various statistical quantities like the mean value of the axial strength. The mean of some variable  $x$  that is measured  $N$  times is estimated as:

$$\mu_x = \frac{1}{N} \sum_{i=1}^N x_i \quad (12.1)$$

Note that this is actually a sample mean, which would differ from the population mean (the true mean if you could measure every bar). With enough samples the sample mean will approach the population mean. In this brief introduction we won't distinguish between sample and population statistics.

Another important quantity is the variance or standard deviation. This is a measure of spread, or how far away our samples are from the mean. The unbiased<sup>†</sup> estimate of the variance is:

$$\sigma_x^2 = \frac{1}{N-1} \sum_{i=1}^N (x_i - \mu_x)^2 \quad (12.2)$$

<sup>†</sup>Unbiased means that the expected value of the sample variance is the same as the true population variance. If  $N$  was used in the denominator, rather than  $N - 1$ , then the two quantities differ by a constant.

and the standard deviation is just the square root of the variance. A small variance implies that measurements are clustered tightly around the mean, whereas a large variance means that measurements are spread out far from the mean. The variance can also be written in the mathematically equivalent, but more computationally friendly format:

$$\sigma_x^2 = \frac{1}{N-1} \left( \sum_{i=1}^N (x_i^2) - N\mu_x^2 \right) \quad (12.3)$$

More generally, we might want to know what the probability is of getting a bar with a specific axial strength. In our testing, we could tabulate the frequency of each measurement in a histogram. If done enough times, it would define a smooth curve as shown in Fig. 12.1a. This curve is called the *probability density function* (PDF),  $p(x)$ , and it tells us the *relative* probability of a certain value occurring. More specifically, a PDF gives the probability of getting a value with a certain range:

$$\text{Prob}[a \leq x \leq b] = \int_a^b p(x)dx \quad (12.4)$$

The total integral of the PDF must be one since it contains all possible outcomes (100%).

$$\int_{-\infty}^{\infty} p(x)dx = 1 \quad (12.5)$$

From the PDF we can also measure various statistics like the mean:

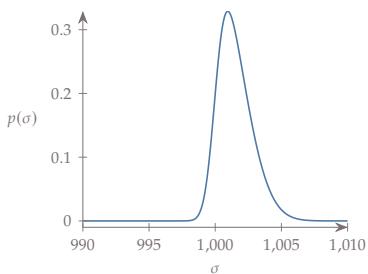
$$\mu_x = E[x] = \int_{-\infty}^{\infty} xp(x)dx \quad (12.6)$$

This quantity is also referred to as the expected value of  $x$  ( $E[x]$ ). We can also compute the variance from its definition:

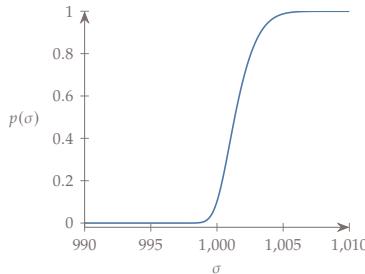
$$\sigma_x^2 = \int_{-\infty}^{\infty} (x - \mu_x)^2 p(x)dx \quad (12.7)$$

or in a mathematically equivalent format:

$$\sigma_x^2 = \int_{-\infty}^{\infty} x^2 p(x)dx - \mu_x^2 \quad (12.8)$$



(a) Probability density function for the axial strength of a rod.



(b) Cumulative distribution function for the axial strength of a rod.

**Figure 12.1:** Comparison between probability density function and cumulative distribution function for a simple example.

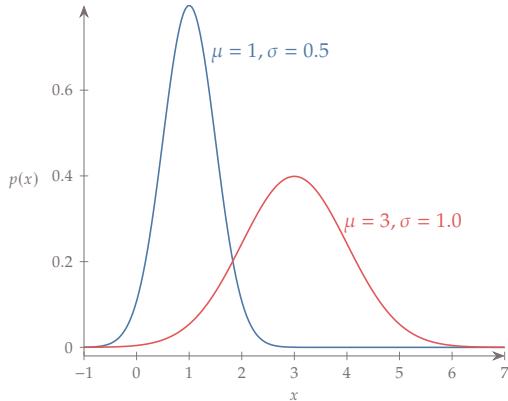
A related concept is the *cumulative distribution function* (CDF), which is the cumulative integral of the PDF:

$$F(x) = \int_{-\infty}^x f(t)dt \quad (12.9)$$

The capital  $F$  denotes the CDF and the lowercase  $f$  the PDF. As an example, the CDF for the axial strength distribution is shown in Fig. 12.1b. The CDF always approaches 1 as  $x \rightarrow \infty$ .

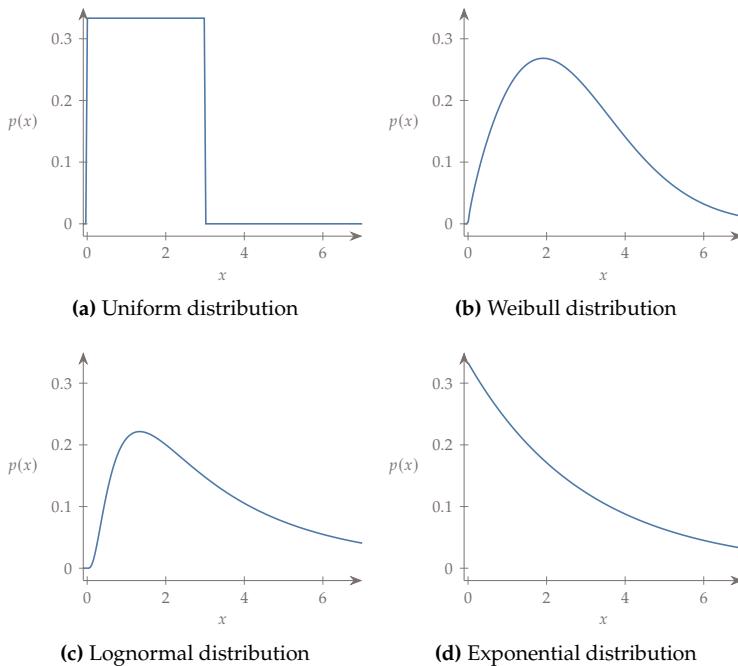
We often fit a named distribution to the PDF of empirical data. One of the most popular distributions is the Gaussian or Normal distribution. Its PDF is:

$$p(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x - \mu)^2}{2\sigma^2} \quad (12.10)$$



**Figure 12.2:** Two normal distributions. Changing the mean causes a shift along the  $x$ -axis. Increasing the standard deviation causes the PDF to spread out.

For a Gaussian distribution the mean and variance are clearly visible in the function, but keep in mind these quantities are defined for any distribution. Figure 12.2 shows two normal distributions with different means and standard deviations to illustrate the effect of those parameters. A few other popular distributions, including a uniform, Weibull, lognormal, and exponential distribution are shown in Fig. 12.3. These only give a flavor of different named distributions, many others exist.



**Figure 12.3:** A few other example probability distributions.

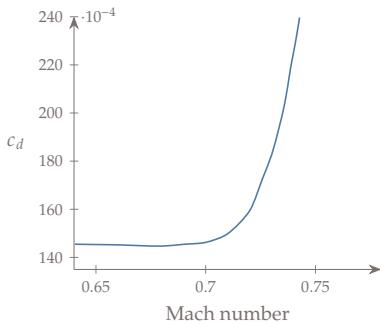
## 12.3 Robust Design

We illustrate some of the key concepts in robust design through examples.

**Example 12.2:** A robust airfoil optimization.

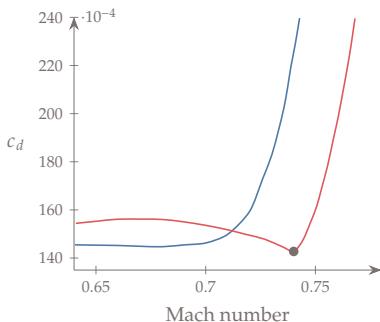
Consider a simple airfoil optimization, with data from the optimization performed by Nemec et al.<sup>133</sup> Figure 12.4 shows the drag coefficient of an RAE 2822 airfoil, as a function of Mach number, evaluated by an inviscid compressible flow solver.

<sup>133</sup>. Nemec et al., *Multipoint and Multi-Objective Aerodynamic Shape Optimization*, 2004



**Figure 12.4:** Inviscid drag coefficient, in counts, of the RAE 2822 airfoil as a function of Mach number.

This is a typical drag rise curve, where increasing Mach number leads to stronger shock waves and an associated increase in wave drag. Now let's try to change the shape of the airfoil to allow us to fly a little bit faster without large increases in drag. We could perform an optimization to minimize the drag of this airfoil at Mach 0.74. The resulting drag curve of this optimized airfoil is shown in Fig. 12.5 in comparison to the baseline RAE 2822 airfoil.

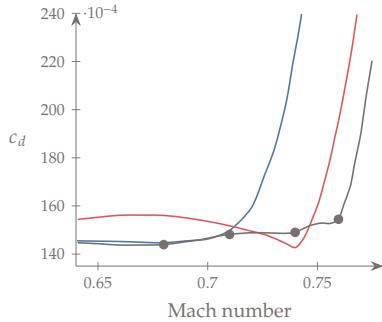


**Figure 12.5:** The red curve shows the drag of the airfoil optimized to minimize drag at  $M = 0.74$ , corresponding to the dot. The drag is low at the requested point, but off-design performance is poor.

Note that the drag is low at Mach 0.74 (as requested!), but any deviation from the target Mach number causes significant drag penalties. In other words, the design is not robust.

One way to improve the design, is to use what is called a multi-point op-

timization. We minimize a weighted sum, equally weighted in this case, of the drag coefficient evaluated at four different Mach numbers ( $M = 0.68, 0.71, 0.74, 0.76$ ). The resulting drag curve is shown in gray in Fig. 12.6.



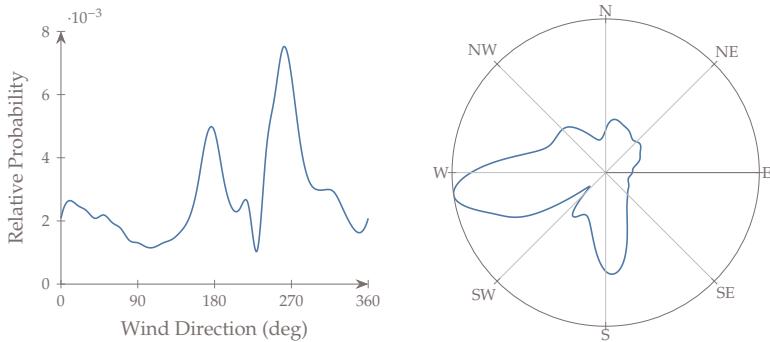
**Figure 12.6:** The gray curve shows the drag of the airfoil optimized to minimize the average drag at the four denoted points. The robustness of the design is greatly improved.

A multipoint optimization is a simplified example of optimization under uncertainty. Effectively, we have treated Mach number as a random parameter with uniform probability across four discrete values. We then minimized the expected value of the drag. This simple change significantly increased the robustness of the design. The drag at our desired speed of Mach 0.74 is not quite as low as the single point case, but it is less sensitive to deviations from this desired operating point. As noted in the introduction, a trade-off in peak performance is required to achieve enhanced robustness.

#### Example 12.3: A robust wind farm layout optimization.

Wind farm layout optimization is another example of optimization under uncertainty, but with a more complex probability distribution compared to the highly simplified multipoint formulation. The positions of wind turbines in a wind farm have a strong impact on overall performance because their wakes interfere with one another. The primary goal of wind farm layout optimization is to position the turbines to reduce interference and thus maximize power production. In this example there are nine turbines, and the constraints on this problem are purely geometric: the turbines must stay within a specified boundary and must not be too close to any other turbine.

One of the primary challenges of wind farm layout optimization is that the wind is highly variable. To keep the example simple, we will assume that wind speed is constant, but that wind direction is an uncertain parameter. Figure 12.7 shows a PDF of wind direction for an actual wind farm, which is known as a wind rose, and is more commonly visualized as shown on the right plot. We see that the wind is predominately out of the west, with another peak coming out of the south. Because of the variable nature of the wind it is difficult to intuit an optimal layout.

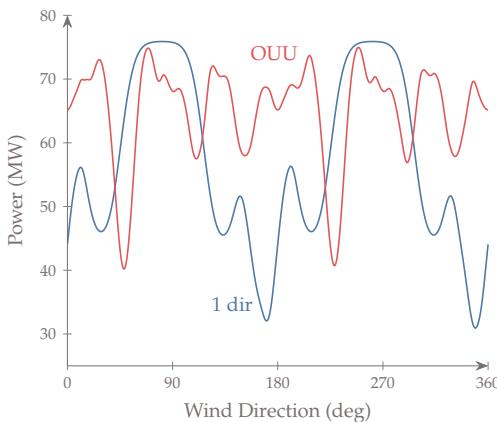


**Figure 12.7:** Left: probability density function of wind direction. Right: same PDF but visualized as a wind rose.

We solve the problem two ways. The first way is to solve the problem deterministically (i.e., ignore the variability). Commonly this is done by using mean values for uncertain parameters, often with the assumption that the variability is Gaussian or at least symmetric. In this case, the wind direction is periodic, and very asymmetric, so instead we optimize using the most probable wind direction ( $261^\circ$ ). The second way is to treat this as an OUU problem. Instead of maximizing the power for one direction, we maximize the expected value of the power across all directions. This is straightforward to compute from the definition of expected value because this is one-dimensional function. Section 12.5 explains other ways to perform forward propagation.

Figure 12.8 shows the power as a function of wind direction for both cases. Note that the deterministic approach does indeed allow for higher power production when the wind comes from the west (and  $180^\circ$  degrees from that), but that power drops considerably for other directions. In contrast, the OUU result is much less sensitive to changes in wind direction. The expected value of power is 58.6 MW for the deterministic case, and 66.1 MW for the OUU case, which represents over a 12% improvement<sup>‡</sup>.

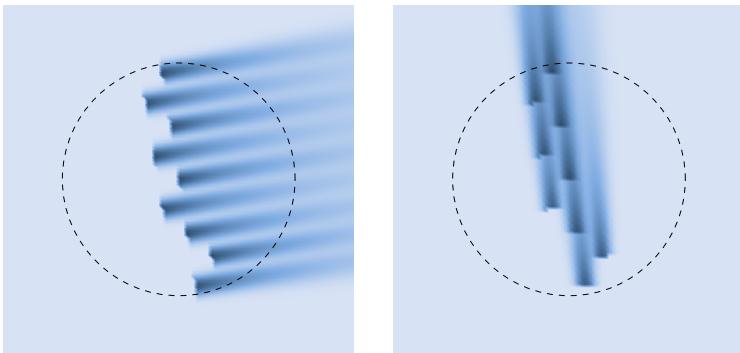
<sup>‡</sup>The wind energy community does not use expected power directly, but rather annual energy production, which is just the expected power times utilization



**Figure 12.8:** Wind farm power, as a function of wind direction, for two cases: optimized deterministically using the most probable direction, optimized under uncertainty.

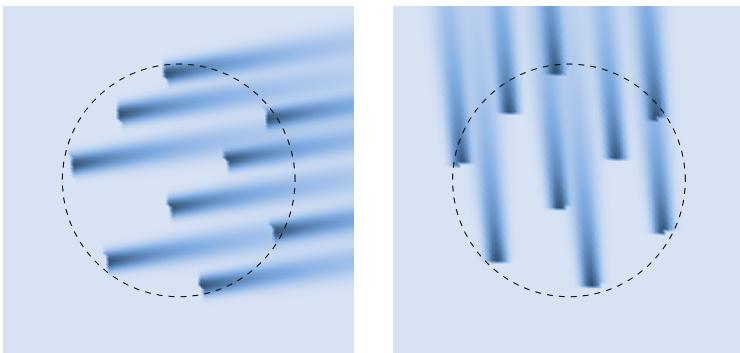
We can also see the tradeoff in the optimal layouts. The left side of Fig. 12.9

shows the optimal layout using the deterministic formulation, with the wind coming from the predominant direction (the direction we optimized for). The wakes are shown in blue and the boundaries with a dashed line. The wind turbines have spaced themselves out so that there is very little wake interference. However, when the wind changes direction the performance degrades significantly. The right side of Fig. 12.9 shows the same layout, but when the wind is in the second-most probable direction. In this direction many of the turbines are operating in the wake of another turbine and produce much less power.



**Figure 12.9:** Left: Deterministic case with the primary wind direction. Right: Deterministic case with the secondary wind direction.

In contrast, the robust design is shown in Fig. 12.10 for the predominant wind direction on the left and the second-most probable direction on the right. In both cases the wake effects are relatively minor, though not quite as ideally placed in the predominant direction. The tradeoff in performance for that one direction, allows the design to be more robust as the wind changes direction.



**Figure 12.10:** Left: OUU case with the primary wind direction. Right: OUU case with the secondary wind direction.

This example again highlights the classic risk-reward tradeoff. The maximum power achieved at the most probable wind speed is reduced, in exchange for reduced power variability and thus higher energy production in the long run.

Both Examples 12.2 and 12.3 used the expected value, or mean, as the objective function. However, there are other useful forms of OUU objective functions that may be more suitable. Consider the following options:

1. Minimize the mean of the function:  $\mu_f(x)$ .
2. Minimize the standard deviation (or variance) of the function:  $\sigma_f(x)$ .
3. Minimize the mean plus or minus some number of standard deviations:  $\mu_f(x) \pm k\sigma_f(x)$ .
4. Perform a multiobjective optimization trading off the mean and standard deviation. A Pareto front can be a useful tool to assess this risk-reward tradeoff (see Chapter 9).
5. Minimize other statistical quantities like the 95% percentile of the distribution.
6. Minimize a reliability metric (see next section for further discussion on reliability):  $\text{Prob}[f(x) > f_{\text{crit}}]$ , which means to minimize the probability that the objective exceeds some critical value.

## 12.4 Reliability

In addition to affecting the objective function, uncertainty also affects constraints. As with robustness, we begin with an example.

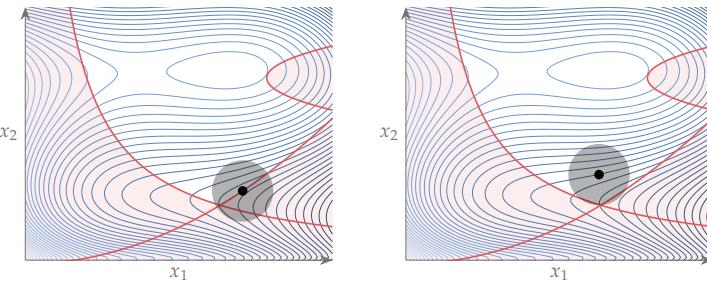
---

**Example 12.4:** Reliability with the Barnes function.

Consider the Barnes function shown on the left side of Fig. 12.11. The three red lines are the three nonlinear constraints of the problem and the red regions highlight regions of infeasibility. With deterministic inputs, the optimal value sits right on the constraint. An uncertainty ellipse shown around the optimal point highlights the fact that the solution is not reliable. Any variability in the inputs will create a significant probability for one or more of the constraints to be violated (just like the real life problem where you are likely to be late if you plan your arrival assuming zero variability).

Conversely, the right side of Fig. 12.11 shows a reliable optimum, with the same uncertainty ellipse. We see that it is highly probable that the design will satisfy all constraints under the input variation. However, as noted in the introduction, increased reliability presents a performance trade-off with a corresponding increase in the objective function.

---



**Figure 12.11:** Left: The constrained deterministic optimum sits right on the constraint and if there is any variability is likely to violate a constraint. Right: The reliable optimum will still satisfy the constraints even with variability.

In some engineering disciplines, increasing the reliability is handled in a basic way through safety factors. These safety factors are deterministic, but are usually derived through statistical means.

---

**Example 12.5:** Connecting safety factors to reliability.

If we were constraining the stress ( $\sigma$ ) in a structure to be less than the material's yield stress ( $\sigma_y$ ) we would not want to use a constraint of the form:

$$\sigma(x) \leq \sigma_y. \quad (12.11)$$

This would be dangerous because we know there is inherent variability in the loads, and uncertainty in the yield stress of the material. Instead we often use a simple safety factor.

$$\sigma(x) \leq \eta\sigma_y, \quad (12.12)$$

where  $\eta$  is a total safety factor that accounts for safety factors from loads, materials, and failure modes. Of course, not all applications have standards-driven safety factors already determined. The statistical approach discussed in this chapter is useful in these situations to allow for reliable designs.

---

To create reliable design we change our deterministic inequality constraints:

$$g(x) \leq 0 \quad (12.13)$$

to the form:

$$\text{Prob}[g(x) \leq 0] \geq R \quad (12.14)$$

where  $R$  is the reliability level. In words, we want the probability of constraint satisfaction to exceed some pre-selected reliability level. For example, if we set  $R = 0.999$  the solution must satisfy the constraints with a probability of 99.9%. Thus, we can explicitly set the reliability level that we wish to achieve, with associated trade-offs in the level of performance for the objective function.

## 12.5 Forward Propagation

In the previous sections we have assumed that we know the statistics (e.g., mean, standard deviation, etc.) of the *outputs* of interest (objectives and constraints). However, we generally do not have that information. Instead, we only know the probability density functions (or some of its magnitudes, e.g., mean, variance) of the *inputs*<sup>§</sup>. Forward propagation methods propagate input uncertainties through a numerical model to compute output statistics.

<sup>§</sup>Or at least we can make some assumptions that characterize the input uncertainties.

Uncertainty quantification is a large field unto itself, and we cannot hope to do more than provide a broad introduction in this chapter. We introduce four well-known methods for forward propagation: direct quadrature, Monte Carlo methods, first-order perturbation methods, and polynomial chaos.

### 12.5.1 Direct Quadrature

The mean and variance of an output function  $f$  are defined as:

$$\mu_f = \int f(x)p(x)dx \quad (12.15)$$

$$\sigma_f^2 = \int f(x)^2 p(x)dx - \mu_f^2 \quad (12.16)$$

and one way to estimate them is to use direct numerical *quadrature*. In other words, the estimation of each integral becomes a summation:

$$\int f(x)dx \approx \sum_i f(x_i)w_i \quad (12.17)$$

where  $w_i$  are specific weights. The nodes where the function is evaluated, and the corresponding weights, are determined by the quadrature strategy (e.g., rectangle rule, trapezoidal rule, Newton–Cotes, Clenshaw–Curtis, Gaussian, Gauss–Kronrod).

The difficulty of numerical quadrature is extending to multiple dimensions (also known as cubature), and, unfortunately, most of the time there is more than one uncertain variable. The most obvious extension for multidimensional quadrature is a full-grid tensor product. This type of grid is created by discretizing the nodes in each dimension, and then evaluating at every combination of nodes. Mathematically, the quadrature formula can be written as

$$\int f(x)dx_1dx_2\dots dx_n \approx \sum_i \sum_j \dots \sum_n f(x_i, x_j, \dots, x_n)w_iw_j\dots w_n \quad (12.18)$$

While conceptually straightforward, this approach is subject to the *curse of dimensionality*. The number of points we need to evaluate at grows exponentially with the number of input dimensions.

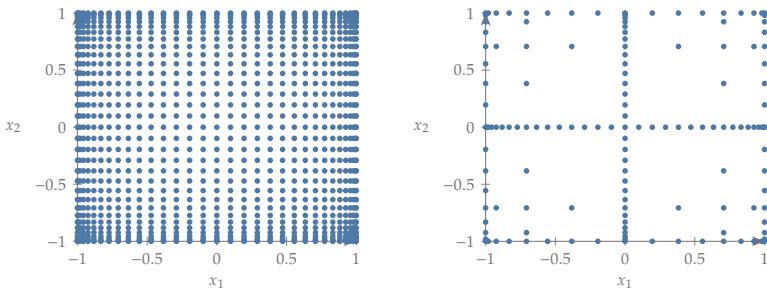
One approach to deal with the exponential growth is to use a sparse grid method first proposed by Smolyak.<sup>134</sup> The details are beyond our scope, but the basic idea is that through intelligently chosen points, we can maintain the level of accuracy of a full tensor grid while evaluating at far fewer points. Different quadrature rules (e.g., trapezoidal, Clenshaw-Curtis) produce different grids.

<sup>134</sup>. Smolyak, *Quadrature and interpolation formulas for tensor products of certain classes of functions*. 1963

---

**Example 12.6:** Sparse grid methods for quadrature.

Figure 12.12 shows a comparison between a two-dimension full tensor grid using the Clenshaw-Curtis exponential rule (left) and a level 5 sparse grid (right) using the same quadrature strategy.



**Figure 12.12:** Comparison between a two-dimensional full tensor grid (left) and a level 5 sparse grid (right) using the Clenshaw-Curtis exponential rule.

For a problem with dimension  $d$ , and approximately  $N$  sample points in each dimension, the full tensor grid has a computational complexity of  $\mathcal{O}(N^d)$ , whereas the sparse grid method has a complexity of  $\mathcal{O}(N(\log N)^{d-1})$  with comparable accuracy. This scaling alleviates the curse of dimensionality to some extent, but the number of evaluation points is still strongly dependent on problem dimensionality—making it intractable in high dimensions. The method introduced in the next section is independent of the problem dimensionality, though is not without its own drawbacks.

### 12.5.2 Monte Carlo simulation

As we saw in the previous section, direct numerical quadrature faces the curse of dimensionality. Monte Carlo simulation can be used to alleviate this issue. Monte Carlo methods attempt to approximate the integrals of the previous section, by using the law of large numbers. The

basic idea is that output probability distributions can be approximated by running the simulation many times with inputs randomly sampled from the input probability distributions. There are three steps:

1. *Random sampling.* Sample  $N$  points  $x_i$  from the input probability distributions using a random number generator.
2. *Numerical experimentation.* Evaluate the outputs at these points:  $f_i = f(x_i)$ .
3. *Statistical analysis.* Compute statistics on the discrete output distribution  $f_i$  (e.g., using Eqs. 12.1 and 12.3).

We can also estimate  $\text{Prob}[c(x) \leq 0]$  by counting how many times the constraint was satisfied and dividing by  $N$ . If we evaluate enough samples, our output statistics will converge to the true values by the law of large numbers (though herein also lies its disadvantage, it requires a *large number* of samples).

The Monte Carlo method has three main advantages. First, as noted in the previous section, the convergence rate is independent of the number of inputs. Whether we have 3 or 300 random input variables, the convergence rate will be similar since we can sample from all inputs at the same time. Second, the algorithm is easy to parallelize since all of the function evaluations are completely independent. Third, in addition to statistics like the mean and variance, we can also generate the output probability distributions.

The major disadvantage of the Monte Carlo method is that even though the convergence rate does not depend on the number of inputs, the convergence rate is slow:  $O(1/\sqrt{N})$ . This means that improving the accuracy by one decimal place requires approximately 100 *times* more samples. That means that if you need three more digits of accuracy you would have to use about *one million times* as many samples. It is also hard to know what value of  $N$  to use *a priori*. Usually we need to determine an appropriate value for  $N$  through convergence testing (trying larger values of  $N$  until the statistics converge).

One approach to achieve converged statistics with fewer iterations is to use LHS instead of pure random sampling. LHS allows one to better approximate the input distributions with fewer samples, and is introduced in Section 10.2, in the context of surrogate modeling. Various related sampling approaches exist, including importance sampling, quasi-Monte Carlo, and Bayesian quadrature. Even with better sampling methods, generally a large number of simulations are re-

quired, which can be particularly prohibitive if used as part of an OUU problem.

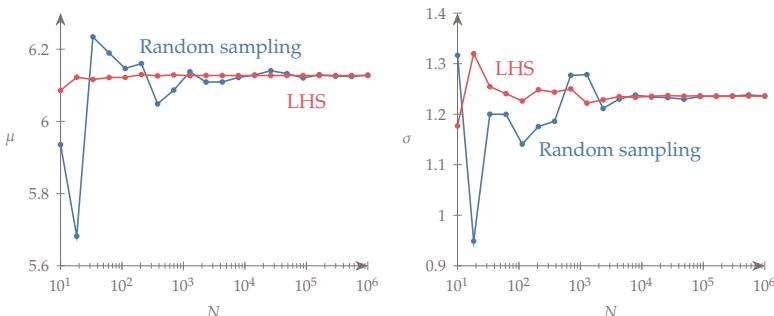
**Example 12.7:** Forward propagation with Monte Carlo

Consider a problem with the following objective and constraint:

$$\begin{aligned} f(x) &= x_1^2 + 2x_2^2 + 3x_3^2 \\ g(x) &= x_1 + x_2 + x_3 \leq 3.5 \end{aligned} \quad (12.19)$$

At the current iteration you are at the point  $x = [1, 1, 1]$ . We assume that the first variable is deterministic, whereas the latter two variables have uncertainty under a normal distribution with the following standard deviations:  $\sigma_2 = 0.06, \sigma_3 = 0.2$ . We would like to compute the outputs statistics for  $f$  (mean, variance, and a histogram), and compute the reliability of the constraint at this current iteration.

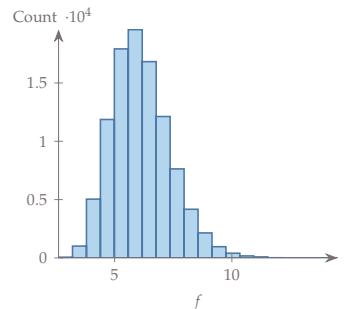
We don't know how many samples we need in order to get reasonably converged statistics, so need to perform a convergence study. For a given number of samples, we generate random numbers normally distributed with mean  $x_i$  and standard deviation  $\sigma_i$ , we then evaluate the functions and compute the mean, variance, and reliability of the outputs. Figure 12.13 shows the convergence of the mean and standard deviation under the "Random sampling" curve.



**Figure 12.13:** Convergence of the mean and standard deviation versus the number of samples using Monte Carlo.

From the data it appears that we may need about  $10^5$  samples to confidently have well converged statistics. Using that number of samples gives the following results:  $\mu = 6.133, \sigma = 1.242$ , reliability = 99.187%. Note that because of the random sampling these results will vary somewhat between simulations. The corresponding histogram of the objective function is seen in Fig. 12.14. The output distribution does not appear to be normally distributed. Producing an output histogram is one of the unique benefits of this method.

As discussed, LHS can be used to converge statistics with fewer samples. The exact same process is repeated, except with LHS rather than pure random sampling. The convergence in the mean and standard deviation is shown under the "LHS" label in Fig. 12.13. We see that convergence is quicker, especially in the mean, requiring fewer samples by at least an order of magnitude.



**Figure 12.14:** Histogram of objective function for 100,000 samples.

---

### 12.5.3 First-order Perturbation Method

Perturbation methods are based on a local Taylor's series expansion of the functional output. If we use a first-order Taylor's series, assume that each uncertain input variable is mutually independent, and assume that the PDFs of  $x_i$  are each symmetric, then we can derive the following expressions for the mean and variance of some output  $f(x)$ :

$$\begin{aligned}\mu_f &= f(\mu_x) \\ \sigma_f^2 &= \sum_{i=1}^n \left( \frac{\partial f}{\partial x_i} \sigma_{x_i} \right)^2\end{aligned}\quad (12.20)$$

where  $x$  contains all the uncertain variables (design variables and parameters).<sup>¶</sup>

The first equation just says the mean value of the function is the function evaluated at the mean value of  $x$ . You may recognize the second equation as it is commonly used in propagating errors from experimental measurements. Its major limitations are 1) it relies on a linearization (first-order Taylor's series) and so will not be as accurate if the local function space is highly nonlinear, 2) it assumes that all uncertain parameters are uncorrelated, which should be true for design variables, but is not necessarily true for parameters, and 3) it assumes symmetry, and so might be too inaccurate for something like the wind farm example (Ex. 12.3).

We have *not* assumed that the input or output distributions are normal (i.e., Gaussian). However, with a first-order series we can only estimate mean and variance and not any of the higher moments (e.g., skewness, kurtosis).

As compared to Monte Carlo or other sampling methods, a perturbation method can be much more efficient, but the end result is not a distribution, but rather just some of its statistics (e.g., mean and variance). Additionally, we see that derivatives appear in our analysis. This is desirable in the sense that the derivatives are what make this method effective and efficient, but it also means that extra information needs to be supplied. If, for example, we were propagating the variance of a constraint for a reliability-based optimization, we would need to supply gradients of the constraints for purposes of Eq. 12.20. Additionally, if using a gradient-based method, we would need derivatives of the whole expression, meaning second derivatives of the constraint with respect to the uncertain variables. If we have exact gradients using the

<sup>¶</sup>Using the covariance matrix allows for a more general expression for first-order perturbation methods.<sup>135</sup> Higher-order Taylor's series can also be used,<sup>136</sup> but they are less common because of their increased complexity.

<sup>135</sup>. Smith, *Uncertainty Quantification: Theory, Implementation, and Applications*, 2013

<sup>136</sup>. Cacuci, *Sensitivity & Uncertainty Analysis, Volume 1*, 2003

methods discussed in Chapter 6 then this is usually not problematic. If not, then propagating the uncertainty inside the analysis can be numerically challenging.

As an alternative, Parkinson et al. proposed a simpler but more approximate alternative for reliability-based optimization where the uncertainty is computed outside of the optimization loop and an additional assumption is made that each constraint is normally distributed.<sup>137</sup> If  $g(x)$  is normally distributed we can rewrite the constraint  $\text{Prob}[g(x) \leq 0] \geq R$  as:

$$g(x) + k\sigma_g \leq 0$$

where  $k$  is chosen for a desired reliability level  $R$ . For example,  $k = 2$  implies a reliability level of 97.72% (one-sided tail of the normal distribution). In many cases an output distribution is reasonably approximated as normal, but for cases with nonnormal output this method can introduce large error.

Keep in mind that with multiple active constraints, one must be careful to appropriately choose the reliability level for each constraint such that the overall reliability is in the desired range. Often the simplifying assumption is made that the constraints are uncorrelated, and thus the total reliability is the product of the reliabilities of each constraint.

This simplified approach has the following steps:

1. Compute the deterministic optimum.
2. Estimate the standard deviation of each constraint  $\sigma_g$  using Eq. 12.20.
3. Adjust the constraints to  $g(x) + k\sigma_g \leq 0$  for some desired reliability level and re-optimize.
4. Repeat as necessary.

While this simplification is approximate, it is very easy to use and the magnitude of error is usually appropriate for the conceptual design phase. If the errors are unacceptable, then the statistical quantities can be computed inside the optimization. Keep in mind that this approach only applies to reliability-based optimization and would not work if there was uncertainty in the objective.

#### 12.5.4 Polynomial Chaos

Polynomial chaos (also known as spectral expansions) is a class of forward propagation methods take advantage of the inherent smoothness

<sup>137</sup> Parkinson et al., *A General Approach for Robust Optimal Design*. 1993

<sup>138</sup> Polynomial chaos is not chaotic and does not actually need polynomials. The name “polynomial chaos” came about because it was initially derived to use in a physical theory of chaos.<sup>138</sup>

<sup>138</sup> Wiener, *The Homogeneous Chaos*. 1938

of the outputs of interest using polynomial approximations.<sup>¶</sup> A general function that depends on uncertain variables  $x$ , can be represented as a sum of basis functions  $\psi_i$  (usually polynomials) with weights  $\alpha_i$ :

$$f(x) = \sum_{i=0}^{\infty} \alpha_i \psi_i(x) \quad (12.21)$$

although in practice we truncate the series after  $n + 1$  terms.

$$f(x) \approx \sum_{i=0}^n \alpha_i \psi_i(x) \quad (12.22)$$

The required number of terms for a given input dimension  $d$ , and polynomial order  $p$  is:

$$n + 1 = \frac{(d + p)!}{d!p!} \quad (12.23)$$

Different types of polynomials are used, but they are always chosen to form an orthogonal basis. Two vectors  $u$  and  $v$  are orthogonal if their dot product is zero

$$\vec{u} \cdot \vec{v} = 0 \quad (12.24)$$

For functions, the definition is similar, but functions have an infinite dimension and so an integral is required instead of a summation. Two functions  $f$  and  $g$  are orthogonal over an interval  $a$  to  $b$  if their inner product is zero. Different definitions for the inner product can be defined. The simplest is:

$$\int_a^b f(x)g(x)dx = 0 \quad (12.25)$$

For our purposes, the weighted inner product is more relevant:

$$\langle f, g \rangle = \int_a^b f(x)g(x)p(x)dx = 0 \quad (12.26)$$

where  $p(x)$  is a weight function, and in our case is the probability density function. The angle bracket notation, known as an inner product, will be used in the remainder of this section.

The intuition is similar to that of vectors. Adding a non-orthogonal vector to a set of vectors does not increase the span of the vector space. In other words, the new vector could have been formed by a linear combination of existing vectors in the set and so does not add any new information. Similarly, we want to make sure that any new basis functions we add are orthogonal to the existing set, so that the range of functions that can be approximated is increased.

You may be familiar with this concept from its use in Fourier series. In fact, this method is a truncated generalized Fourier series. Recall that a Fourier series represents an arbitrary periodic function with a series of sinusoidal functions. The basis functions in the Fourier series are orthogonal.

By definition we choose the first basis function to be  $\psi_0 = 1$ . This just means the first term in the series is a constant (polynomial of order 0). Because the basis functions are orthogonal we know that

$$\langle \psi_i, \psi_j \rangle = 0 \text{ if } i \neq j \quad (12.27)$$

Three main steps are involved in using these methods:

1. Select a orthogonal polynomial basis.
2. Compute coefficients to fit the desired function.
3. Compute statistics on the function of interest.

These three steps are overviewed below, though we begin with the last step because it provides insight for the first two.

### Compute Statistics

Using Eq. 12.6 the mean of the function  $f$  is given by:

$$\mu_f = \int f(x)p(x)dx \quad (12.28)$$

where  $p(x)$  is the PDF for  $x$ . Using our polynomial approximation we can express this as:

$$\mu_f = \int \sum_i \alpha_i \psi_i(x)p(x)dx \quad (12.29)$$

The coefficients  $\alpha_i$  are constants and so can be taken out of the integral:

$$\begin{aligned} \mu_f &= \sum_i \alpha_i \int \psi_i(x)p(x)dx \\ &= \alpha_0 \int \psi_0(x)p(x)dx + \alpha_1 \int \psi_1(x)p(x)dx + \alpha_2 \int \psi_2(x)p(x)dx + \dots \end{aligned}$$

Recall that  $\psi_0 = 1$ . Also, because we can multiply all terms by 1 without changing anything, we can rewrite this expression in terms of our defined inner product as:

$$\mu_f = \alpha_0 \int p(x)dx + \alpha_1 \langle \psi_0, \psi_1 \rangle + \alpha_2 \langle \psi_0, \psi_2 \rangle + \dots \quad (12.30)$$

Because the polynomials are orthogonal, all of the terms except the first are zero (see Eq. 12.27), and by definition of a PDF, we know that the integral of  $p(x)$  over the domain must be one (see Eq. 12.5). Thus, we have the simple result that the mean of the function is simply given by the zeroth coefficient:

$$\mu_f = \alpha_0 \quad (12.31)$$

Using a similar approach we can derive a formula for the variance. By definition, the variance is (Eq. 12.8):

$$\sigma_f^2 = \int f(x)^2 p(x) dx - \mu_x^2 \quad (12.32)$$

Substituting our polynomial representation and using the same techniques used in deriving the mean:

$$\begin{aligned} \sigma_f^2 &= \int \left( \sum_i \alpha_i \psi_i(x) \right)^2 p(x) dx - \alpha_0^2 \\ &= \sum_i \alpha_i^2 \int \psi_i(x)^2 p(x) dx - \alpha_0^2 \\ &= \alpha_0^2 \int \psi_0^2 p(x) dx + \sum_{i=1}^n \alpha_i^2 \int \psi_i(x)^2 p(x) dx - \alpha_0^2 \\ &= \alpha_0^2 + \sum_{i=1}^n \alpha_i^2 \int \psi_i(x)^2 p(x) dx - \alpha_0^2 \\ &= \sum_{i=1}^n \alpha_i^2 \int \psi_i(x)^2 p(x) dx \\ &= \sum_{i=1}^n \alpha_i^2 \langle \psi_i^2 \rangle \end{aligned}$$

The inner product  $\langle \psi_i^2 \rangle = \langle \psi_i, \psi_i \rangle$  can generally be computed analytically, and if not, it can be computed easily through quadrature because it only involves the basis functions. For multiple uncertain variables, the formulas are the same:

$$\mu_f = \alpha_0 \quad (12.33)$$

$$\sigma_f^2 = \sum_{i=1}^n \alpha_i^2 \langle \Psi_i^2 \rangle \quad (12.34)$$

except that  $\Psi_i$  are multidimensional basis polynomials defined by products of single dimensional polynomials as will be shown in the next

section. The main takeaway is that the polynomial chaos formulation allows us to compute the mean and variance easily, using our definition of the inner product, and other statistics can be estimated by sampling the polynomial expansion.

### Selecting an Orthogonal Polynomial Basis

Referring again to Eq. 12.26 we need to find polynomials that satisfy the orthogonality relationship for a particular probability density function. For some continuous probability distributions, corresponding orthogonal polynomials are already known.<sup>\*\*</sup> Table 12.1 summarizes the polynomials that correspond to some common probability distributions.<sup>139</sup> Referring again to Eq. 12.26, the polynomials correspond to  $f(x)$  and  $g(x)$ , the probability distribution is  $p(x)$ , and the support range forms the limits of integration  $a$  and  $b$ . For a general distribution, e.g., one that was empirically derived, the orthogonal polynomials should be generated numerically to preserve exponential convergence.<sup>139</sup>

<sup>\*\*</sup> Actually other polynomials can be used, but these particular choices are optimal as they produce exponential convergence.

<sup>139</sup> Eldred et al., *Evaluation of Non-Intrusive Approaches for Wiener-Askey Generalized Polynomial Chaos*. 2008

<sup>139</sup> Eldred et al., *Evaluation of Non-Intrusive Approaches for Wiener-Askey Generalized Polynomial Chaos*. 2008

**Table 12.1:** Orthogonal polynomials that correspond to some common probability distributions.

Prob. Distribution	Polynomial	Support Range
Normal	Hermite	$[-\infty, \infty]$
Uniform	Legendre	$[-1, 1]$
Beta	Jacobi	$[-1, 1]$
Exponential	Laguerre	$[0, \infty]$
Gamma	Generalized Laguerre	$[0, \infty]$

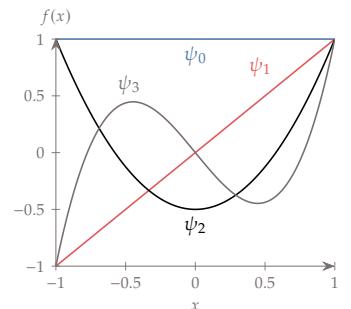
---

**Example 12.8:** Legendre polynomials.

The first few Legendre polynomials are:

$$\begin{aligned} \psi_0 &= 1 \\ \psi_1 &= x \\ \psi_2 &= \frac{1}{2}(3x^2 - 1) \\ \psi_3 &= \frac{1}{2}(5x^3 - 3x) \\ &\vdots \end{aligned} \tag{12.35}$$

These polynomials are plotted in Fig. 12.15, and are orthogonal with respect to a uniform probability distribution.



**Figure 12.15:** The first few Legendre polynomials.

Multidimensional basis functions are simply defined by tensor products. For example, if we had two variables from a uniform probability distribution (and thus Legendre bases), then the first few polynomials, up through second-order terms, are:

$$\begin{aligned}\Psi_0(x) &= \psi_0(x_1)\psi_0(x_2) = 1 \\ \Psi_1(x) &= \psi_1(x_1)\psi_0(x_2) = x_1 \\ \Psi_2(x) &= \psi_0(x_1)\psi_1(x_2) = x_2 \\ \Psi_3(x) &= \psi_1(x_1)\psi_1(x_2) = x_1x_2 \\ \Psi_4(x) &= \psi_2(x_1)\psi_0(x_2) = \frac{1}{2}(3x_1^2 - 1) \\ \Psi_5(x) &= \psi_0(x_1)\psi_2(x_2) = \frac{1}{2}(3x_2^2 - 1)\end{aligned}$$

Note that  $\psi_1\psi_2$ , for example, does not appear in the above list because that is a third order polynomial and we truncated the series at second-order terms. We should expect this number of basis function because Eq. 12.23, with  $d = 2, p = 2$ , suggests that we should have six basis functions.

### Determine Coefficients

With the polynomial basis  $\psi_i$  fixed, we need to determine the appropriate coefficients  $\alpha_i$  in Eq. 12.22. We discuss two ways to do this. The first is with quadrature and is also known as non-intrusive spectral projection. The second is with regression and is also known as stochastic collocation.

First, let's use the quadrature approach. Beginning with the polynomial approximation:

$$f(x) = \sum_i \alpha_i \psi_i(x) \quad (12.36)$$

we take the inner product of both sides with respect to  $\psi_j$ .

$$\langle f(x), \psi_j \rangle = \sum_i \alpha_i \langle \psi_i, \psi_j \rangle \quad (12.37)$$

Making use of the orthogonality of the basis functions (Eq. 12.27) we see that all of the terms in the summation are zero except:

$$\langle f(x), \psi_i \rangle = \alpha_i \langle \psi_i^2 \rangle \quad (12.38)$$

or

$$\alpha_i = \frac{1}{\langle \psi_i^2 \rangle} \int f(x) \psi_i(x) p(x) dx \quad (12.39)$$

Note that, as expected, the zeroth coefficient is simply the definition of the mean.

$$\alpha_0 = \int f(x)p(x)dx \quad (12.40)$$

The remaining coefficients must be obtained through multidimensional quadrature using the same types of approaches as discussed in Section 12.5.1 (or even Section 12.5.2). This means that this approach inherits the same limitations of the chosen quadrature approach, though the process can be more efficient if the distributions are well approximated by the selected basis functions.

It may appear that to estimate  $f(x)$  (Eq. 12.22) we need to know  $f(x)$  (Eq. 12.39). The distinction is that we just need to be able to evaluate  $f(x)$  and some predefined quadrature points, which in turn gives a polynomial approximation for all of  $f(x)$ .

The second approach to determining the coefficients is regression. The equation in Eq. 12.22 is a linear equation and so we can estimate the coefficients using least squares (although an underdetermined system with regularization can be used as well). This means that we evaluate the function  $m$  times, with  $x^{(i)}$  denoting the  $i^{\text{th}}$  sample, resulting in a linear system:

$$\begin{bmatrix} \psi_0(x^{(1)}) & \dots & \psi_n(x^{(1)}) \\ \vdots & & \vdots \\ \psi_0(x^{(m)}) & \dots & \psi_n(x^{(m)}) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} f(x^{(1)}) \\ \vdots \\ f(x^{(m)}) \end{bmatrix} \quad (12.41)$$

Generally, we would like  $m$ , the number of sample points, to be at least twice as large as  $n + 1$ , the number of unknowns. Determining where to sample, also known as the collocation points, is an important element for this procedure to be effective.<sup>††</sup>

## 12.6 Summary

Engineering problems are subject to variation under uncertainty. Optimizing when design variables and/or parameters have uncertain variability is called optimization under uncertainty. Robust optimization seeks designs that are less sensitive to inherent variability in the objective function. Common OUU objectives include minimizing the mean or standard deviation, or performing multiobjective tradeoffs in mean and standard deviation. Reliable optimization seeks designs that have a reduced probability of failure due to variability in the constraints. In both scenarios, we need methods that propagate probability distributions of the inputs (e.g., design variables) to statistics and in

<sup>††</sup>Several packages exist to facilitate use of polynomial chaos methods.<sup>140,141</sup>

140. Adams *et al.*, *Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.0 User's Manual*. 2015

141. Feinberg *et al.*, *Chaospy: An open source tool for designing methods of uncertainty quantification*. 2015

some cases probability distributions of the outputs (e.g., objective and constraints). This procedure is called forward propagation.

Four classes of forward propagation were discussed in this chapter.<sup>#</sup> Direct quadrature uses numerical quadrature to evaluate the mean and variance of outputs. This process is relatively straightforward and effective. Its primary weakness is that it is limited to low dimensional problems (number of stochastic inputs). Sparse grids extend the dimensional range, but not greatly.

Monte Carlo methods approximate the integrals (and output distributions) using random sampling and the law of large numbers. These methods are extremely easy to use and are independent of problem dimension. Their major weakness is that they are inefficient, though many of the alternatives are completely intractable at high dimensions, making this an appropriate choice for many high-dimensional problems.

Perturbation methods use a Taylor's series expansion of the output functions to estimate the mean and variance. These methods can be efficient across a range of problem sizes, especially if accurate derivatives are available. Their main weaknesses are that they require derivatives (and hence second derivatives if using a gradient-based optimization method), only work with symmetric input probability distributions, and only provide the mean and variance (for first-order methods).

Polynomial chaos represents uncertain variables as a sum of orthogonal basis functions. This method is often a more efficient way to characterize both statistical moments as well as output distributions. However, the methodology is more complex, and is generally limited to a small number of dimensions as the number of required basis functions required grows exponentially.

## Problems

12.1 Answer *true* or *false* and justify your answer.

- Optimizing under uncertainty yields an objective value that is no better than without considering uncertainty.
- Optimization under uncertainty considers uncertainty in the design variables as well as uncertainty in the models that compute the objective and constraint functions.
- The greater the reliability, the less likely the design is to have a worse objective function value.

<sup>#</sup>This list is not exhaustive. For example, all of these methods discussed in this chapter are non-intrusive. Like intrusive methods for derivative computation, intrusive methods for forward propagation require more upfront work but are more accurate and efficient.

- d) Reliability can be handled in a deterministic way using safety factors, which ensure that the optimum has some margin before the original constraint is violated.
- e) Robust design can be obtained by simultaneously optimizing a design for multiple conditions without using uncertainty propagation.
- f) Forward propagation computes the probability density functions of the outputs and inputs for a given numerical model.
- g) The computational cost of direct quadrature scales exponentially with the number of random variables, while the cost Monte Carlo is independent of the number of random variables.
- h) Monte Carlo methods approximate probability density functions using random sampling and converge slowly.
- i) The first-order perturbation method computes the probability density functions using local Taylor's series expansions.
- j) Since the first-order perturbation method requires first-order derivatives to compute the uncertainty metrics, optimization under uncertainty using the first-order perturbation method requires second-order derivatives.
- k) Polynomial chaos is a forward propagation technique that uses polynomial approximations with random coefficients to model the input uncertainties.
- l) The number of basis functions required by polynomial chaos grows exponentially with the number of uncertain input variables.

12.2 *Simplified reliability-based optimization.* This problem uses the simplified version of a first-order perturbation method (Section 12.5.3). The optimization problem below is a QP for simplicity, but ignore that structure and solve it as a general nonlinear problem so that you could reuse the approach (i.e., use a general nonlinear optimizer and a method for estimating gradients).

For the following problem:

$$\begin{aligned} \text{minimize } & f = x_1^2 + 2x_2^2 + 3x_3^2 \\ \text{subject to } & g_1 : 2x_1 + x_2 + 2x_3 \geq 6 \\ & g_2 : -5x_1 + x_2 + 3x_3 \leq -10 \end{aligned}$$

- a) Find the deterministic optimum.

- b) Find the worst-case, reliable optimum where  $\Delta x_1 = \Delta x_2 = \pm 0.1, \Delta x_3 = \pm 0.05$
- c) Now, instead of the worst case tolerances, assume the variables are normally distributed with  $\sigma_{x_1} = \sigma_{x_2} = \pm 0.033, \sigma_{x_3} = \pm 0.0167$  (these values are  $\sigma_i = \Delta_i/3$ ). Find the reliable optimum where the target constraint reliability is 99.865% for each constraint individually.
- d) Compare the total target reliability with a Monte Carlo simulation of reliability for all three approaches (using the normal distributions for the input variations).
- e) Briefly discussed any lessons learned.

### 12.3 Robust optimization of a wind farm.

We want to find the optimal turbine layout for a wind farm in order to minimize cost of energy (COE). We will consider a very simplified wind farm with only three wind turbines. The first turbine will be fixed at  $(0, 0)$  and the  $x$ -positions of the back two turbines will be fixed with 4 diameter spacing between them. The only thing we can change is the  $y$ -position of the two back turbines as shown in Fig. 12.16 (all dimensions in this problem are in terms of rotor diameters). In other words, we just have two design variables:  $y_2$  and  $y_3$ .

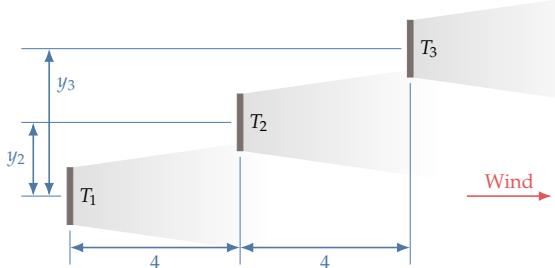


Figure 12.16: Wind farm layout

We further simplify by assuming the wind always comes from the west as shown in the figure, and is always at a constant speed. The wake model has a few parameters that define things like its spread angle and decay rate. We will refer to this parameters as  $\alpha, \beta$ , and  $\delta$  (knowing exactly what each parameter corresponds to is not important for our purposes). The supplementary resources repository contains code for this problem.

- a) Run the optimization deterministically. In other words we will assume that the three wake parameters are deterministic:

$\alpha = 0.1$ ,  $\beta = 9$ ,  $\delta = 5$ . Because there are several possible similar solutions we will add the following constraints:

$$y_i > 0 \quad (\text{bound})$$

$$y_3 > y_2 \quad (\text{linear})$$

Don't use  $[0, 0]$  as the starting point for the optimization, as that occurs right at a flat spot in the wake (a fixed point) and so you might not make any progress. Report the optimal spacing that you find.

- b) Now the wake parameters are not deterministic, but are rather uncertain variables under some probability distribution. Specifically we have the following information for the three parameters:

- $\alpha$  is governed by a Weibull distribution with a scale parameter of 0.1, and a shape parameter of 1.
- $\beta$  is given by a Normal distribution with a mean and standard deviation of  $\mu=9$ ,  $\sigma=1$
- $\delta$  is given by a Normal distribution with a mean and standard deviation of  $\mu=5$ ,  $\sigma=0.4$

Note that the mean for all of these distributions corresponds to the deterministic value we used previously.

Using a Monte Carlo method, run an optimization under uncertainty minimizing the 95th percentile for COE.

- c) Once you have completed both optimizations, you should perform a cross analysis by filling out the four numbers in the table below. You take the two optimal designs that you found, and you compare each on the two objectives (deterministic, and 95th percentile). The first row corresponds to the performance of the optimal deterministic layout. Evaluate the performance of this layout using the deterministic value for COE, and the 95th percentile that accounts for uncertainty. Repeat for the optimal solution for the OUU case. Discuss your findings.

	Deterministic COE	95th percentile COE
Deterministic Layout	[ ]	[ ]
OUU Layout	[ ]	[ ]

As mentioned in Chapter 1, most engineering systems are multidisciplinary, which means that the system is composed of different disciplines that are coupled. We prefer the term *component* instead of “discipline” because it is more general, but we use both terms interchangeably depending on the context. When components in a system represent different physics, the term “multiphysics” is commonly used.

All the optimization methods covered so far apply to multidisciplinary problems if we view the coupled multidisciplinary model as just another model that computes the objective and constraint functions for a given set of design variables. However, there are additional considerations in the solution, derivative computation, and optimization of coupled systems.

In this chapter, we build on Chapter 3 by introducing coupled models and solvers for coupled systems. We also expand the derivative computation methods of Chapter 6 to the coupled system case. Finally, we introduce various MDO *architectures*, which are different options for formulating and solving an MDO problem.

By the end of this chapter you should be able to:

1. Describe when and why one might want to use different optimization architectures.
2. Understand the differences between monolithic and distributed architectures.
3. Read XDSM diagrams.
4. Understand how derivatives are computed in coupled models.

## 13.1 Motivation

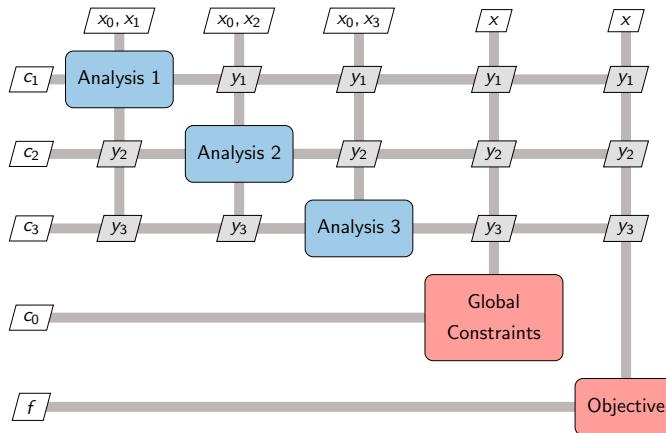
\*

\*Kroo<sup>142</sup> describes many of the early challenges for large-scale MDO and some proposed solutions

<sup>142</sup>. Kroo, *MDO for Large-Scale Design*. 1997

## 13.2 MDO Problem Representation

The MDO problem representation we use here is shown in Fig. 13.1 for a general three-component system. Here we use the functional representation introduced in Section 13.3.2, where the states in each component are hidden and we just see its output as a coupling variable at the system level.



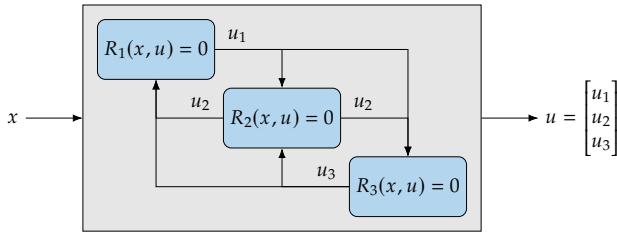
**Figure 13.1:** MDO problem nomenclature and dependencies.

In MDO problems, we make the distinction between *local* design variables, which directly affect only one component, and *global* design variables, which directly affect more than one component. We denote the vector of design variables local to component  $i$  by  $x_i$  and global variables by  $x_0$ . The full vector of design variables is given by  $x = [x_0^T, x_1^T, \dots, x_N^T]^T$ .

The set of constraints is also split into global constraints and local ones. Local constraints are computed by the corresponding component and depend only on the variables available in that component. Global constraints depend on more than one set of coupling variables. These dependencies are also shown in Fig. 13.1.

## 13.3 Multidisciplinary Models

In general, these models would be coupled as shown in Fig. 13.2 for a three-component case. Here, the states of each component affect all other components, but it is common for a component to depend only on a subset of the other system components.

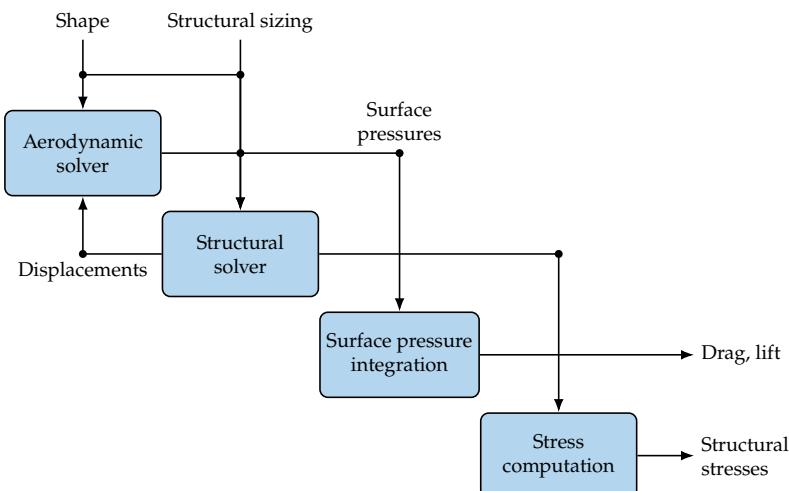


**Example 13.1:** Defining a multidisciplinary optimization problem.

Consider a multidisciplinary numerical model for an aircraft wing, where the aerodynamics and structures disciplines are coupled to solve an aerostructural analysis and design optimization problem. For a given flow condition, the aerodynamic solver computes the forces on the wing for a given wing shape, while the structural solver computes the wing displacement for a given set of applied forces. Thus, these two disciplinary models are coupled as shown in Fig. 13.3. For a steady flow condition, there is only one wing shape and a corresponding set of forces that satisfies both disciplinary models simultaneously.

One possible design optimization problem based on these models would be to minimize the drag by changing the wing shape and the structural sizing, while satisfying a lift constraint and structural stress constraints. In this case, the structural sizing variables are local design variables because only the structural solver is directly affected by those variables. However, the wing shape variables are global design variables because they affect both the aerodynamics and the structure.

**Figure 13.2:** Multidisciplinary model composed of three numerical models. Each model solves for its own state variable vector ( $u_1, u_2, u_3$ ) but in general requires the other state vectors as inputs. This set of models would replace the single model in Fig. 3.17.



**Figure 13.3:** Multidisciplinary numerical model for an aircraft wing.

Mathematically, a multidisciplinary model is no more than a larger

set of equations to be solved, where all the governing equation residuals ( $r$ ), the corresponding state variables ( $u$ ), and all the design variables ( $x$ ) are concatenated into single vectors. Then, we can still just write the whole multidisciplinary model as  $r(x, u) = 0$ .

However, it is often necessary or advantageous to partition the system into smaller components for three main reasons. First, specialized solvers are often already in place for a given set of governing equations, which may be more efficient at solving their set of equations than a general-purpose solver. In addition, some of these solvers might be black boxes that do not provide an interface for using alternative solvers. Second, there is an incentive for building the multidisciplinary system in a modular way. For example, a component might be useful on its own and should therefore be usable outside the multidisciplinary system. A modular approach also facilitates the extension of the multidisciplinary system and makes it easy to replace the model of a given discipline with an alternative one. Finally, the overall system of equations may be more efficiently solved if it is partitioned in a way that exploits the system structure (see Section 13.5.4).

### 13.3.1 Components

In Section 3.3, we explained how all models can ultimately be written as a system of residuals,  $r(x, u) = 0$ . When the system is large or includes sub-models, it might be natural to *partition* the system into *components*. We prefer to use the more general term components instead of disciplines to refer to the sub-models resulting from the partitioning because the partitioning of the overall model is not necessarily by discipline (e.g., aerodynamics, structures). A system model might also be partitioned by physical system component (e.g., wing, fuselage, or an aircraft in a fleet) or by different conditions applied to the same model (e.g., aerodynamic analyses at different flight conditions).

The partitioning can also be performed within a given discipline for the same reasons cited above. In theory, the system model equations in  $r(x, u) = 0$  can be partitioned in any way, but only some partitions are advantageous or make sense. The partitioning can also be hierarchical, where a given component has one or multiple levels of sub-components. Again, this might be motivated by efficiency, modularity, or both.

We denote a partitioning into  $n$  components as

$$r(u) = 0 \Leftrightarrow \begin{cases} r_1(u_1, \dots, u_i, \dots, u_n) = 0 \\ \vdots \\ r_i(u_1, \dots, u_i, \dots, u_n) = 0 \\ \vdots \\ r_n(u_1, \dots, u_i, \dots, u_n) = 0 \end{cases}, \quad (13.1)$$

where each  $r_i$  and  $u_i$  are *vectors* corresponding to the residuals and states of component  $i$ . Here, we assume that each component can drive its residuals to zero by varying only its states, although this is not guaranteed in general. In the above, we have omitted the dependency on  $x$  because for now, we are just concerned with finding the state variables that solve the governing equations for a fixed design. A generic example with three components was illustrated in Fig. 13.2.

Components can be either implicit or explicit, a concept we introduced in Section 3.3. To solve an implicit component, we need an algorithm for driving the equation residuals,  $r_i(u_1, \dots, u_i, \dots, u_n) = 0$ , to zero by varying the states  $u_i$ , while the other states remain fixed. This algorithm could involve a matrix factorization in the case of a linear system or a Newton solver for the case of a nonlinear system. An explicit component is much easier to solve because the state of the component is an explicit function of the states of the other components  $u_i = g(u_j)$  for all  $j \neq i$ . This can be computed without factorization or iteration. There is no loss of generality with the residual notation above because the explicit component can be written as,

$$r_i(u_1, \dots, u_n) = u_i - g(u_j) = 0 \quad \forall j \neq i. \quad (13.2)$$

Most disciplines involve a mix of implicit and explicit components because, as mentioned in Section 3.3 and shown in Fig. 3.17, the state variables are implicitly defined, while the objective function and constraints are usually explicit functions of the state variables. In addition, a discipline usually includes functions that translate inputs and outputs, as discussed next.

---

**Example 13.2:** Residuals of the coupled aerostructural problem.

Let us formulate model for the aerostructural problem described in Ex. 13.1. A possible model for the aerodynamics is a lifting line model given by the linear system,

$$A\Gamma = \gamma, \quad (13.3)$$

where  $A$  is the matrix of aerodynamic influence coefficients and  $\gamma$  is a vector, both of which depend on the wing shape. The state  $\Gamma$  is a vector that represents

the circulation (vortex strength) at each spanwise position on the wing. The lift and drag scalars can be computed explicitly for a given  $\Gamma$ , so we will write these dependencies as  $L = L(\Gamma)$  and  $D = D(\Gamma)$ , omitting the detailed explicit expressions for conciseness.

A possible model for the structures is a cantilevered beam modeled with Euler–Bernoulli elements,

$$Kd = f, \quad (13.4)$$

where  $K$  is the stiffness matrix, which depends on the beam shape and sizing. The right-hand-side vector represents the applied forces at spanwise position on the beam. The states  $d$  are the displacements and rotations of each element. The weight does not depend on the states and it is an explicit function of the beam sizing and shape, so it does not involve the structural model (13.4). The stresses are an explicit function of the displacements, so we can write  $\sigma = \sigma(d)$ , where  $\sigma$  is a vector whose size is the number of elements.

When we couple these two models,  $A$  and  $\gamma$  depend on the wing displacements  $d$ , and  $f$  depends on the  $\Gamma$ . We can write all the implicit and explicit equations as residuals:

$$\begin{aligned} r_1 &= A(d)\Gamma - \gamma(d) = 0 \\ r_2 &= L - L(\Gamma) = 0 \\ r_3 &= D - D(\Gamma) = 0 \\ r_4 &= Kd - f(\Gamma) = 0 \\ r_5 &= \sigma - \sigma(d) = 0. \end{aligned} \quad (13.5)$$

We used Eq. (13.2) to transform the explicit equations into residuals for  $r_2$ ,  $r_3$ , and  $r_5$ . The states of this system are,

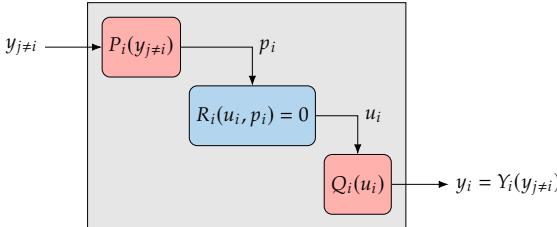
$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} \equiv \begin{bmatrix} \Gamma \\ L \\ D \\ d \\ \sigma \end{bmatrix}. \quad (13.6)$$

Because  $u_2$ ,  $u_3$ , and  $u_5$  can be explicitly determined from  $u_1$  and  $u_4$ , we could just solve for  $r_1$  and  $r_4$  and then just use the explicit expressions. However, it can be convenient to write all equations as a singular vector  $r(u) = 0$ .

### 13.3.2 System-level representations and coupling variables

In MDO, *coupling variables* ( $y$ ) are variables that need to be passed from one component to the other due to interdependencies in the system. Sometimes, the coupling variables are just the state variables of one component (or a subset of these) that get passed to another component. In the general case for a component  $i$ , there is an intermediate explicit function ( $P_i$ ) that translates the inputs coming from the other

components ( $y_{j \neq i}$ ) to the required inputs  $p_i$ , as shown in Fig. 13.4. After the component solves for its state variables  $u_i$ , there might be another function ( $Q_i$ ) that converts these states to output variables for other components. The function ( $Q_i$ ) typically reduces the number of output variables relative to the number of internal states, some times by orders of magnitude.



**Figure 13.4:** In the general case, a solver includes a conversion of inputs and outputs distinct from its states.

The *system level* representation of a coupled system is determined by the variables that are “seen” and controlled at this level. These variables are the system level variables that the system level solver is responsible for. If the box shown in Fig. 13.4 is viewed as a black box, then the internal states  $u_i$  would be hidden at the system level, and the relationship between its inputs and outputs can be represented by a single function as  $y_i = Y_i(y_{j \neq i})$ . We call this the *functional* representation of a coupled system. If a component is a black box and we have no access to the residuals and the translation functions, this is the only representation we get to see. This functional representation can be written as

$$y = Y(y) \Leftrightarrow \begin{cases} y_1 = Y_1(y_2, \dots, y_n) \\ \vdots \\ y_i = Y_i(y_{j \neq i}) \\ \vdots \\ y_n = Y_n(y_1, \dots, y_{n-1}) \end{cases}, \quad (13.7)$$

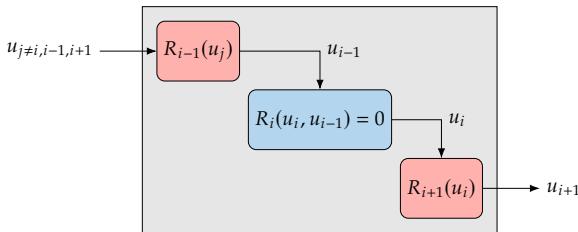
In this representation, we do not use the component residuals associated with the state variables, so we need some other residuals associated with the coupling variables. Thus, the residuals of the functional representation can be written as

$$R_i = y_i - Y_i(y_{j \neq i}) = 0, \quad (13.8)$$

where  $y_i$  are the guesses for the coupling variables and  $Y_i$  are the actual computed values.

The *residual* representation of the coupled system is an alternative system level representation, where a general component, including

the translation functions, is represented by a set of residuals and corresponding states, i.e.,  $R(u) = 0$ , as written earlier in Eq. 13.1. This residual form is desirable because as we will see later in this chapter, this enables us to formulate an efficient general way of solving coupled systems and computing their derivatives. A system can be converted to residuals and states by converting the functions (which are explicit) to residuals using Eq. 13.2. The result is a component with three subcomponents as shown in Fig. 13.5. This hints at another powerful concept that we will use later, which is the concept of hierarchy, where components can contain sub-components and multiple levels are present.



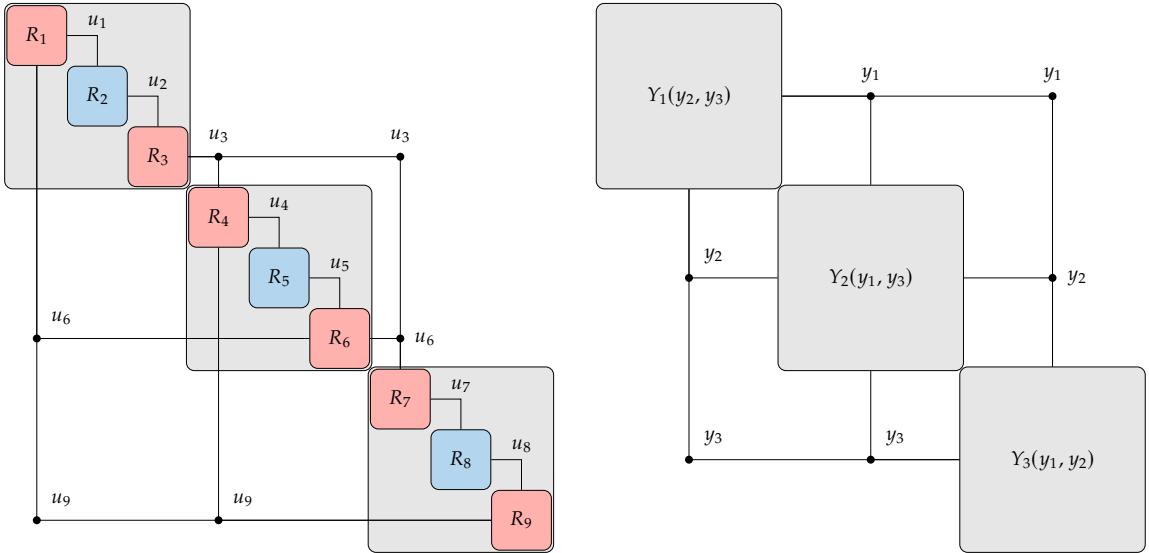
**Figure 13.5:** The translation of inputs and outputs can be represented as components with its own state variables, so any coupled system can be written as  $R(u) = 0$ .

An example of a system with three solvers is shown in Fig. 13.6. On the left figure, we show the three solvers written as a set of residuals and states, including the translation functions. Each solver in this general case requires three components to represent it. In the case where the solver is a black box, the governing equations, residuals, and translation functions are hidden, and all we see are the coupling variables. In an even more general case, these two views can be mixed, where some solvers have exposed residuals and states, while others do not. Furthermore, there might be translation functions between black boxes that are exposed.

### 13.3.3 Coupled System Representations

To show how multidisciplinary systems are coupled, we use a design structure matrix (DSM), sometimes referred to as a dependency structure matrix or an  $N^2$ -diagram. An example of the DSM for a hypothetical system is shown in Fig. 13.7a. In this matrix, the diagonal elements represent the components, while the off-diagonal entries denote coupling variables. A given coupling variable is computed by the component in its row and is passed to the component in its column.<sup>†</sup> As shown in Fig. 13.7a, there are in general off-diagonal entries both above and below the diagonal, where the entries above feed forward, while entries below feed backward.

<sup>†</sup>In some of the DSM literature this definition is reversed, where “row” and “column” are interchanged, resulting in a transposed matrix.

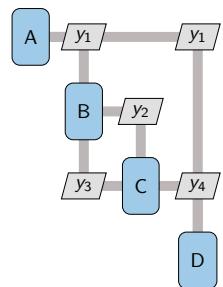


The mathematical representation of these dependencies is given by a graph (Fig. 13.7b), where the graph nodes are the components and the edges represent the information dependency. This graph is a *directed graph* because in general there are three possibilities for a coupling: a single coupling one way or the other, or a two way coupling. A directed graph is said to be *cyclic* when there are edges that form a closed loop, or cycles. In the example of Fig. 13.7b, there is a single cycle between components B and C. When there are no closed loops, the graph is *acyclic*. In this case, the whole system can be solved by solving each component in turn, without having to iterate. A graph can also be represented using an *adjacency matrix* (Fig. 13.7c), which has the same structure as the transpose of the DSM.

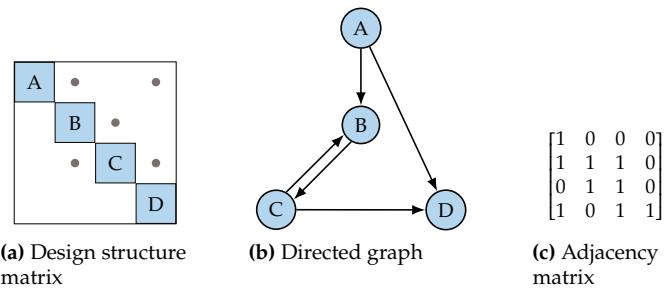
The adjacency matrix for real-world systems is often a *sparse matrix*, that is, it has many zeros in its entries. This means that in the corresponding DSM, each component depends only on a subset of all the other components. We can take advantage of the structure of this sparsity in the solution of coupled systems.

The DSM shows only data dependencies. We now introduce an extended version of the DSM, called XDSM, which we use later in this chapter to show *process* in addition to the data dependencies. Fig. 13.8 shows the XDSM for the same four-component system. When showing only the data dependencies, the only difference relative to DSM is that the coupling variables are labeled explicitly, and the data paths are drawn. In the next section, we add process to the XDSM.

**Figure 13.6:** Two system-level views of coupled system with three solvers: all components exposed and written as residuals and states (left) and black box representation where only inputs and outputs for each solver are visible (right), where  $y_1 \triangleq u_3$ ,  $y_2 \triangleq u_6$ , and  $y_3 \triangleq u_9$ .



**Figure 13.8:** XDSM showing data dependencies for the four-component coupled system of Fig. 13.7.



**Figure 13.7:** Different representations of the dependencies of a hypothetical system.

### 13.3.4 Solving Coupled Numerical Models

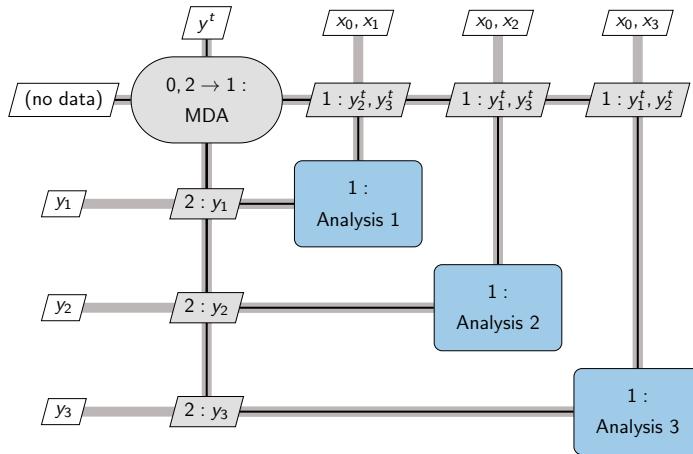
When considering the solution of coupled systems, also known as multidisciplinary analysis (MDA), we usually assume that a solver already exists that determines the states for each component and the coupling variables it provides to the coupled system. Thus, in the *system-level view*, we only deal with the coupling variables (denoted as  $y$ ) and the internal states ( $u$ ) are hidden.

The most straightforward way to solve for coupled numerical models is through a fixed-point iteration, which is analogous to the fixed-point iteration methods mentioned in Section 3.7 and detailed in Appendix B. The difference here is that instead of updating one state at the time, we update a vector of coupling variables at each iteration corresponding to a subset of the coupling variables in the overall coupled system. Obtaining this vector of coupling variables in general involves the solution of a nonlinear system. Therefore, these are called *nonlinear block* variants of the linear fixed-point iteration methods.

The nonlinear block Jacobi method requires a guess for all coupling variables to start with and calls for the solution of all components given those guesses. Once all components have been solved, the coupling variables are updated based on the new values computed by the components, and all components are solved again. This iterative process continues until the coupling variables do not change in subsequent iterations. Because each component takes the coupling variables values from the previous iteration, which have already been computed, all components can be solved in parallel without communication. This algorithm is formalized in Alg. 13.3. When applied to a system of components, we call it the block-Jacobi method, where “block” refers to each component.

The nonlinear block Jacobi method is also illustrated using an XDSM in Fig. 13.9 for three components. The only input is the guess for the coupling variables,  $u^t$ . The MDA block (step 0) is responsible for iterating the system-level analysis loop and for checking if the system

has converged. The process line is shown as the thin black line to distinguish from the data dependency connections (thick gray lines), and follows the sequence of numbered steps. The analyses for each component are all numbered the same (step 1), because they can be done in parallel. Each component returns the coupling variables it computes to the MDA iterator, closing the loop between step 2 and step 1 (denoted as  $2 \rightarrow 1$ ).



**Figure 13.9:** A nonlinear block Jacobi multidisciplinary analysis process to solve a three-component coupled system.

---

### Algorithm 13.3: Nonlinear block Jacobi algorithm

---

**Inputs:**
 $u^{(0)} = [u_1^{(0)}, \dots, u_n^{(0)}]$ : *Guesses for coupling variables*
**Outputs:**
 $u = [u_1, \dots, u_n]$ : *System-level states*


---

```

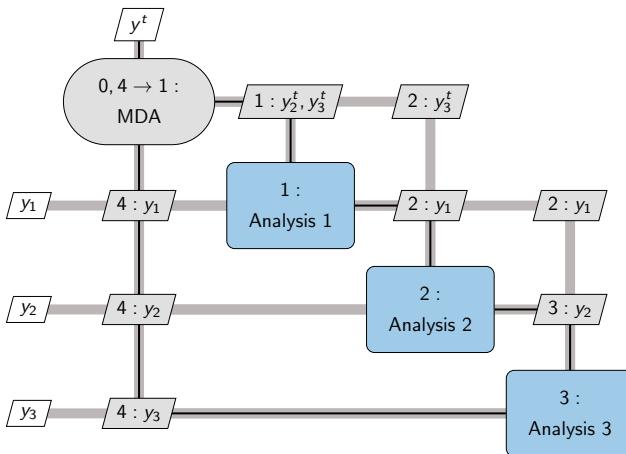
 $k = 1$ 
while  $\|u^{(k)} - u^{(k-1)}\|_2 < \epsilon$  do
    for all  $i \in \{1, \dots, n\}$  do
         $u_i^{(k)} \leftarrow \text{solve } R_i(u_i^{(k)}, u_j^{(k-1)}) = 0, \text{ where } j \neq i$  Can be done in parallel
    end for
     $k = k + 1$ 
end while

```

---

The nonlinear block Gauss–Seidel algorithm is similar to its Jacobi counterpart. The only difference is that when solving each component, we use the latest coupling variables available instead of just using

the coupling variables from the previous iteration. We cycle through each component  $i = 1, \dots, n$  in order. When computing  $u_i$  by solving component  $i$  at iteration  $k$ , for all  $u_j$  that are inputs to component  $i$ , we use  $u_j^{(k)}$  for all  $j < i$  and  $u_j^{(k-1)}$  for all  $j > i$ . This results in a better convergence rate in general, but the components cannot be solved in parallel because now each component depends on the current iteration's coupling variables from all previous components in the sequence. This algorithm is illustrated in Fig. 13.10 and formalized in Alg. 13.4.



**Figure 13.10:** A block Gauss–Seidel multidisciplinary analysis (MDA) process to solve a three-discipline coupled system.

---

**Algorithm 13.4:** Nonlinear block Gauss–Seidel algorithm

---

**Inputs:**

$u^{(0)} = [u_1^{(0)}, \dots, u_n^{(0)}]$ : *Guesses for coupling variables*

**Outputs:**

$u = [u_1, \dots, u_n]$ : *System-level states*

---

```

 $k = 1$ 
while  $\|u^{(k)} - u^{(k-1)}\|_2 < \epsilon$  do
    for  $i = 1, n$  do
         $u_i^{(k)} \leftarrow \text{solve } R_i(u_1^{(k)}, \dots, u_{i-1}^{(k)}, u_i^{(k)}, u_{i+1}^{(k-1)}, \dots, u_n^{(k-1)}) = 0$ 
    end for
     $k = k + 1$ 
end while

```

---

Both the block Jacobi and Gauss–Seidel methods converge linearly, but Gauss–Seidel converges more quickly because each equation uses

the latest information available.

The order in which the components are solved makes a big difference in the efficiency of the Gauss–Seidel method. In the best possible scenario, the components can be reordered such that there are no entries in the lower diagonal of the DSM, which means that each component depends only on previously solved components and there are therefore no feedback dependencies. In this case the block Gauss–Seidel method would converge to the solution in one forward sweep.

In the more general case, even though we might not be able to completely eliminate the lower diagonal entries, minimizing these entries by reordering results in better convergence. This reordering can also make the difference between convergence and non-convergence.

Newton's method can also be applied to the system level. For a system of nonlinear residual equations, the Newton step in the coupling variables,  $\Delta u = u^{(k+1)} - u^{(k)}$  can be found by solving the linear system

$$\frac{\partial R}{\partial u} \Big|_{u=u^{(k)}} \Delta u = -R \left( u^{(k)} \right), \quad (13.9)$$

where we need the partial derivatives of all the residuals with respect to the coupling variables to form the Jacobian matrix  $\partial R / \partial u$ .

Expanding the concatenated residual and coupling variable vectors, we get,

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \dots & \frac{\partial R_1}{\partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial u_1} & \dots & \frac{\partial R_n}{\partial u_n} \end{bmatrix} \begin{bmatrix} \Delta u_1 \\ \vdots \\ \Delta u_n \end{bmatrix} = - \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix}, \quad (13.10)$$

where the derivatives in the block Jacobian matrix and the right hand side are evaluated at the current iteration,  $u^{(k)}$ . These derivatives can be computed using any of the methods from Chapter 6. Note that this Jacobian matrix has exactly the same structure of the DSM and is often a sparse matrix. The full procedure is listed in Alg. 13.5.

---

**Algorithm 13.5:** Newton method for system-level convergence

---

**Inputs:**

$$u^{(0)} = [u_1^{(0)}, \dots, u_n^{(0)}]: \text{Guesses for coupling variables}$$

**Outputs:**

$$u = [u_1, \dots, u_n]: \text{System-level states}$$


---

$k = 1$

**while**  $\|R\|_2 < \epsilon$  **do**

**for all**  $i \in \{1, \dots, n\}$  **do**

*Can be done in parallel*

```

Compute  $R_i$ 
Compute  $\frac{\partial R_i}{\partial u_j}$  for  $j = 1, \dots, n$ 
end for
 $\Delta u \leftarrow$  solve block Newton system (13.10)
 $u^{(k+1)} = u^{(k)} + \Delta u$ 
 $k = k + 1$ 
end while

```

---

Like the plain Newton method, this coupled-Newton method has similar advantages and disadvantages. The main advantage is that it converges quadratically once it is close enough to the solution (if the problem is well conditioned). The main disadvantage is that it might not converge at all, depending on the initial guess.

### 13.4 Coupled Derivative Computation

As we well know by now, gradient-based optimization requires the derivatives of the objective and constraints with respect to the design variables. Any of the methods for computing derivatives from Chapter 6 can be used, but some require modifications. The difference is that, in MDO, the computation of the functions of interest (objective and constraints) requires the solution of a coupled system of components.

The finite-difference method can be used with no modification, as long as an MDA is converged well enough for each perturbation in the design variables. As explained in Section 6.4, the cost of computing derivatives with the finite-difference method is proportional to the number of variables. The constant of proportionality can increase significantly compared to that of a single discipline because the MDA convergence might be slow (especially if using a Jacobi or Gauss–Seidel iteration).

The precision of the derivatives depends directly on the precision of the functions of interest. In previous sections, we only needed to concern ourselves with the precision of a single component. Now that the function of interest depends on a coupled system, the precision of the MDA must be considered. This precision depends on the precision of each component as well as the convergence of the MDA. Even if each component provides precise function values, the precision of the derivatives might be compromised if the MDA is not converged well enough.

The complex-step method and forward mode AD can also be used for a coupled system, but some modifications are required. The complex-

step method requires all components to be able to take complex input and compute the corresponding complex outputs, and similarly, AD requires inputs and outputs that include derivative information. For a given MDA, if one of these methods are applied to each component and the coupling includes the derivative information, we can compute the derivatives of the coupled system. When using AD, manual coupling will be required if the components and the coupling are programmed in different languages. While both of these methods produce precise derivatives for each component, the precision of the derivatives for the coupled system could be compromised by a low level of convergence of the MDA. The reverse mode of AD for coupled systems would be more involved: After an initial MDA, a reverse MDA would be run to compute the derivatives.

Analytic methods (both direct and adjoint) can also be extended to compute the derivatives of coupled systems. All the equations derived for a single component in Section 6.7 are valid for coupled system if we concatenate the residuals and the state variables.

Thus, the coupled version of the linear system for the direct method (6.45) is

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \dots & \frac{\partial R_1}{\partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial u_1} & \dots & \frac{\partial R_n}{\partial u_n} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} \frac{\partial R_1}{\partial x} \\ \vdots \\ \frac{\partial R_n}{\partial x} \end{bmatrix}, \quad (13.11)$$

where  $\phi_i$  is the derivatives of the states from component  $i$  with respect to the design variable. Once we have solved for  $\phi$ , we can use the coupled equivalent of the total derivative equation (6.46) to compute the derivatives.

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \left[ \frac{\partial F}{\partial u_1}, \dots, \frac{\partial F}{\partial u_n} \right] \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_n \end{bmatrix}. \quad (13.12)$$

The coupled adjoint equations can be written as

$$\begin{bmatrix} \frac{\partial R_{1T}}{\partial u_1} & \dots & \frac{\partial R_{1T}}{\partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_{nT}}{\partial u_1} & \dots & \frac{\partial R_{nT}}{\partial u_n} \end{bmatrix}^T \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial u_1} \\ \vdots \\ \frac{\partial F}{\partial u_n} \end{bmatrix}^T. \quad (13.13)$$

After solving for the coupled-adjoint vector using the equation above, we

can use the total derivative equation to compute the desired derivatives,

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - [\psi_1^T, \dots, \psi_n^T] \begin{bmatrix} \frac{\partial R_1}{\partial x} \\ \vdots \\ \frac{\partial R_n}{\partial x} \end{bmatrix}. \quad (13.14)$$

There is an alternative form for the coupled direct and adjoint methods that was not useful for single models. The coupled direct and adjoint methods derived above use the residual form of the governing equations and are a natural extension of the corresponding methods applied to single models. In this form, the residuals for all the equations and the corresponding states are exposed at the system level. As previously mentioned in Section 13.3.2, there is an alternative system-level representation—the functional representation—that views each model as a function relating its inputs and outputs  $y = Y(y)$ , where the coupling variables  $y$  represent the system-level states. We can derive direct and adjoint methods from the functional representation.

The functional versions of these methods can be derived by defining the residuals as  $R(y) \triangleq y - Y(y) = 0$ , where the states are now the coupling variables. The linear system for the direct method (13.11) then yields

$$\begin{bmatrix} I & -\frac{\partial Y_1}{\partial y_2} & \dots & -\frac{\partial Y_1}{\partial y_n} \\ \vdots & \ddots & \ddots & \vdots \\ -\frac{\partial Y_n}{\partial y_1} & -\frac{\partial Y_n}{\partial y_2} & \dots & I \end{bmatrix} \begin{bmatrix} \bar{\phi}_1 \\ \vdots \\ \bar{\phi}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial Y_1}{\partial x} \\ \vdots \\ \frac{\partial Y_n}{\partial x} \end{bmatrix}, \quad (13.15)$$

where  $\bar{\phi}_i = dy_i/dx$ . The total derivatives of the function of interest can then be computed with

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - \left[ \frac{\partial F}{\partial y_1}, \dots, \frac{\partial F}{\partial y_n} \right] \begin{bmatrix} \bar{\phi}_1 \\ \vdots \\ \bar{\phi}_n \end{bmatrix}. \quad (13.16)$$

The functional form of the coupled adjoint equations can be similarly derived, yielding

$$\begin{bmatrix} I & -\frac{\partial Y_1 T}{\partial y_2} & \dots & -\frac{\partial Y_1 T}{\partial y_n} \\ \vdots & \ddots & \ddots & \vdots \\ -\frac{\partial Y_n T}{\partial y_1} & -\frac{\partial Y_n T}{\partial y_2} & \dots & I \end{bmatrix}^T \begin{bmatrix} \bar{\psi}_1 \\ \vdots \\ \bar{\psi}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial Y_1}{\partial x} \\ \vdots \\ \frac{\partial Y_n}{\partial x} \end{bmatrix}, \quad (13.17)$$

After solving for the coupled-adjoint vector using the equation above, we can use the total derivative equation to compute the desired derivatives,

$$\frac{df}{dx} = \frac{\partial F}{\partial x} - [\bar{\psi}_1^T, \dots, \bar{\psi}_n^T] \begin{bmatrix} \frac{\partial F}{\partial y_1} \\ \vdots \\ \frac{\partial F}{\partial y_n} \end{bmatrix}. \quad (13.18)$$

Finally, the unification of the methods for computing derivatives introduced in Section 6.9 also applies to coupled systems and can be used to derive the coupled direct and adjoint methods presented above. Furthermore, the UDE (6.74) can also handle residual or functional components, as long as they are ultimately expressed as residuals.

---

**Tip 13.6:** Coupled derivative computation.

Obtaining derivatives for each component of a multidisciplinary model and propagating them to compute the coupled derivatives usually requires a high implementation effort. OpenMDAO<sup>‡</sup> was designed to help with this problem. Even if exact derivatives can only be supplied for a subset of the models and the rest are obtained by finite difference, the system derivatives will usually be more accurate than applying finite difference at the system level.

<sup>‡</sup><http://openmdao.org>

## 13.5 Monolithic Architectures

Monolithic MDO architectures cast the design problem as a single optimization. The only difference between the different monolithic architectures is the set of design variables that the optimizer is responsible for, which has repercussions on the set of constraints considered and how the governing equations are solved.

§

<sup>§</sup>Martins *et al.*<sup>34</sup> present a comprehensive description of all MDO architectures, including references to known applications of each architecture.

### 13.5.1 Multidisciplinary Feasible

The multidisciplinary design feasible (MDF) architecture is the closest to a single discipline problem because the design variables, objective, and constraints are the same as we would expect for a single discipline problem. The only difference is that the computation of the objective and constraints requires the solution of a coupled system instead of a single system of governing equations. Therefore, all the optimization

algorithms covered in the previous chapters can be applied without modification in MDF. The resulting optimization problem is

$$\begin{aligned}
 & \text{minimize} && f(x, y^*) \\
 & \text{by varying} && x \\
 & \text{subject to} && c_0(x, y^*) \leq 0 \\
 & && c_i(x_0, x_i, y_i^*) \leq 0, \quad i = 1, \dots, N. \\
 & \text{while solving} && R_i(x, y) = 0 \quad i = 1, \dots, N. \\
 & \text{by varying} && y
 \end{aligned} \tag{13.19}$$

where an MDA is performed for each  $x$  solve for the internal component states and the coupling variables  $y^*$ , using any of the methods from Section 13.3.4. An MDA is converged when it finds a set of coupling variables  $y^*$  that makes all components consistent. That is, computing the output coupling variables for each component using  $y^*$  as an input,

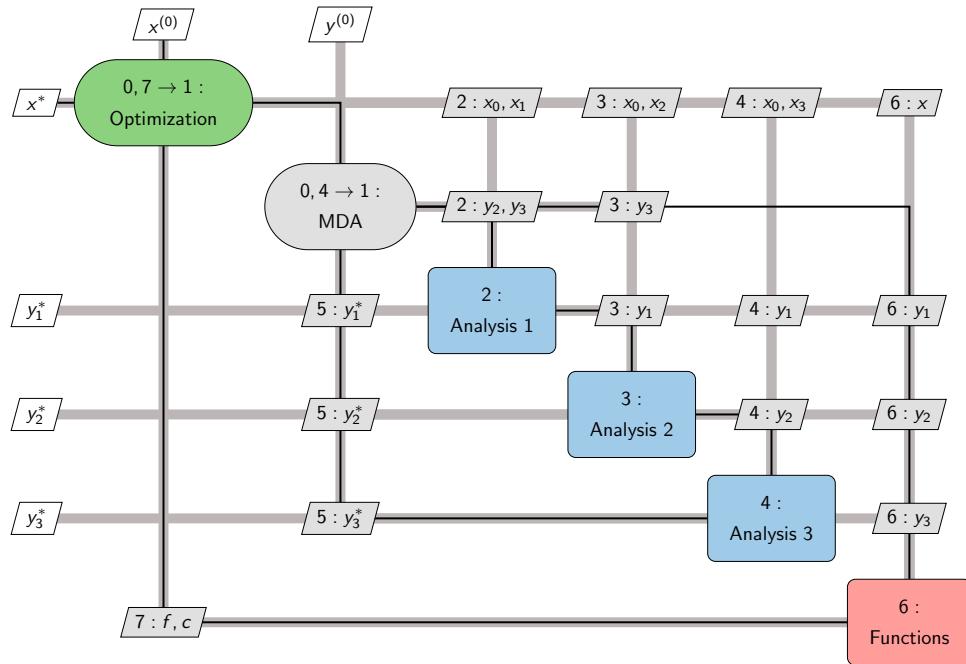
$$y_i = Y_i(x_0, x_i, y_{j \neq i}^*), \quad i = 1, \dots, N, \tag{13.20}$$

yields  $y_i = y_i^*$  for all components. Then, the objective and constraints can be computed based on the current design variables and coupling variables. An XDSM for MDF with three components is shown in Fig. 13.11. Here we use a Gauss–Seidel iteration to converge the MDA, but any other method for converging the MDA could be used.

One advantage of MDF is that the system-level states are physically compatible if an optimization stops prematurely. This is advantageous in an engineering design context when time is limited and we are not as concerned with finding an optimal design in the strict mathematical sense as with finding an improved design. However, it is not guaranteed that the design constraints are satisfied if the optimization is terminated early; that depends on whether the optimization algorithm maintains a feasible design point or not.

The main disadvantage of MDF is that it requires an MDA for each optimization iteration, which requires its own algorithm outside of the optimization. Implementing an MDA algorithm can be time consuming if one is not already in place. One of the easiest to implement is the block Gauss–Seidel algorithm, but as we have seen, it converges slowly.

When using a gradient-based optimizer, gradient calculations are also challenging for MDF because it requires coupled derivatives. Finite-different derivative approximations are easy to implement, but their poor scalability and precision are compounded by the MDA, as explained in Section 13.4. Ideally, we would use one of the analytic coupled derivative computation methods of Section 13.4, which require a substantial implementation effort.



### 13.5.2 Individual Discipline Feasible

The individual discipline feasible (IDF) architecture adds independent copies of the coupling variables to allow component analyses to run independently and possibly in parallel. These copies are known as *target variables*, are controlled by the optimizer, while the actual coupling variables are computed by the corresponding component. Target variables are denoted by a superscript  $t$  so that the coupling variables produced by discipline  $i$  is  $y_i^t$ . These variables represent the current guesses for the coupling variables that are independent of the corresponding actual coupling variables computed by each component. To ensure the eventual consistency between the target coupling variables and the actual coupling variables at the optimum, we define a set of *consistency constraints*,  $c_i^c = y_i^t - y_i$ , which we add to the optimization problem formulation.

**Figure 13.11:** The MDF architecture relies on an MDA to solve for the coupling and state variables for each optimization iteration. In this case, the MDA uses a Gauss-Seidel approach.

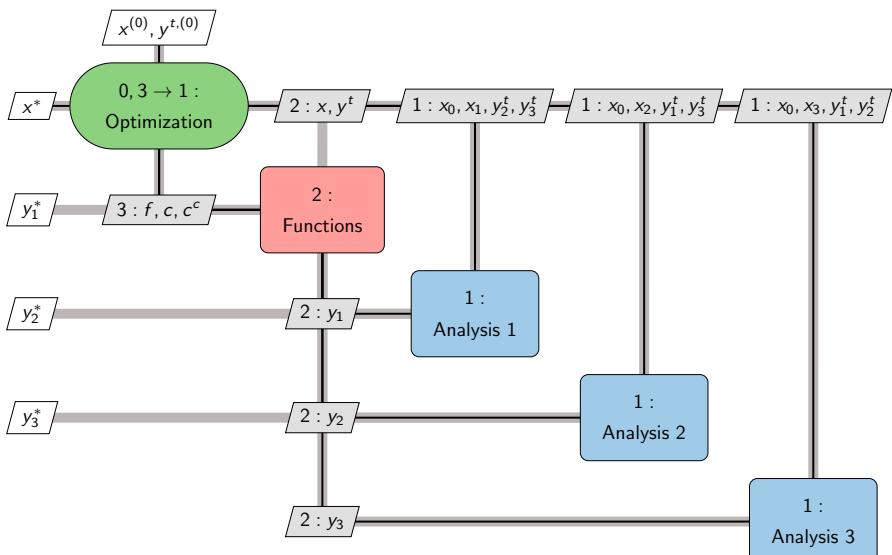
The optimization problem for the IDF architecture is

$$\begin{aligned}
 & \text{minimize} \quad f(x, y) \\
 & \text{by varying} \quad x, y^t \\
 & \text{subject to} \quad c_0(x, y) \leq 0 \\
 & \quad c_i(x_0, x_i, y_i) \leq 0 \quad i = 1, \dots, N, \\
 & \quad c_i^c = y_i^t - y_i = 0 \quad i = 1, \dots, N, \\
 & \text{while solving} \quad R_i(x, y_i, y_{j \neq i}^t) = 0 \quad i = 1, \dots, N. \\
 & \text{for} \quad y
 \end{aligned} \tag{13.21}$$

where each component is solved independently to compute the corresponding output coupling variables and constraints based on the target coupling variables, that is

$$\begin{aligned}
 y_i &= Y_i(x_0, x_i, y_{j \neq i}^t), \\
 c_i &= c_i(x_0, x_i, y_i, y_{j \neq i}^t),
 \end{aligned} \tag{13.22}$$

where unlike MDF, we do not need to make the coupling variables consistent using an MDA. Instead, each component is solved independently for each optimization iteration. Then  $f$  and  $c_0$  are computed using the current design variables  $x$  and the latest available set of coupling variables  $y$ . The XDSM for IDF is shown in Fig. 13.12.



**Figure 13.12:** The IDF architecture breaks up the MDA by letting the optimizer solve for the coupling variables that satisfy interdisciplinary feasibility.

One advantage of IDF is that each component can be solved in parallel because they do not depend on each other directly. Instead, the coupling between the components is resolved by the optimizer, which iterates the target coupling variables,  $y^t$  until it satisfies the consistency constraints,  $c^c$ , such that  $y^t = y$ .

This leads to the main disadvantage of IDF, which is that the optimizer must handle more design variables and constraints compared to the MDF architecture. If the number of coupling variables is large, the size of the resulting optimization problem might be too large to solve efficiently. This problem can be mitigated by careful selection of the components or by aggregating the coupling variables to reduce their dimensionality.

Another advantage of IDF is that if a gradient-based optimization algorithm is used to solve the optimization problem, the optimizer is typically more robust and has better convergence rate than the fixed-point iteration algorithms of Section 13.3.4.

### 13.5.3 Simultaneous Analysis and Design

Simultaneous analysis and design (SAND) extends the idea of IDF by moving not only the coupling variables to the optimization problem, but all component states as well. The SAND approach requires exposing all the components in form of the system-level view previously introduced in Fig. 13.6. The residuals of the analysis become constraints that the optimizer is responsible for.<sup>¶</sup>

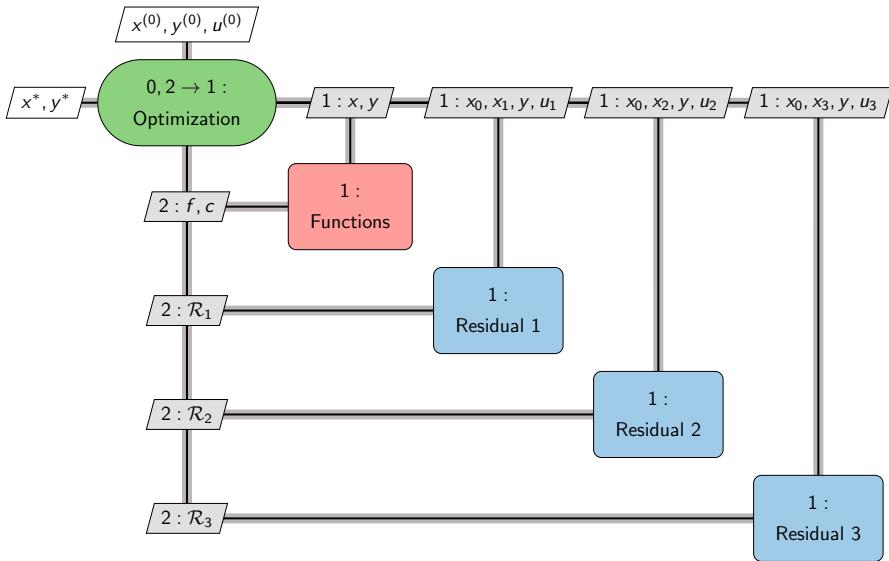
This means that component solvers are no longer needed and the optimizer becomes responsible for simultaneously solving the components for their states, the interdisciplinary compatibility for the coupling variables, and the design optimization problem for the design variables. Because the optimizer is controlling all these variables, SAND is also known as a full-space approach. SAND can be stated as

$$\begin{aligned} & \text{minimize } f_0(x, y) \\ & \text{by varying } x, y, u \\ & \text{subject to } c_0(x, y) \leq 0 \\ & \quad c_i(x_0, x_i, y_i) \leq 0 \quad \text{for } i = 1, \dots, N \\ & \quad R_i(x_0, x_i, y, u_i) = 0 \quad \text{for } i = 1, \dots, N. \end{aligned} \tag{13.23}$$

where we use the representation shown in Fig. 13.4, and therefore there are two sets of explicit functions that translate the input coupling variables of the component. The SAND architecture is also applicable to single components, in which case there are no coupling variables. The XDSM for SAND is shown in Fig. 13.13

<sup>¶</sup>When the residual equations arise from discretized PDEs, we have what is called *PDE-constrained optimization*. <sup>143</sup>

<sup>143</sup> Biegler *et al.*, *Large-Scale PDE-Constrained Optimization*. 2003



Because we are solving all variables simultaneously, the SAND architecture has the potential for being the most efficient way to get to the optimal solution. In practice, however, it is unlikely that this is advantageous when efficient component solvers are available.

The resulting optimization problem is the largest of all MDO architectures and requires an optimizer that scales well with the number of variables. Therefore, a gradient-based optimization algorithm is likely required, in which case, the derivative computation must also be considered. Fortunately, SAND does not require derivatives of the coupled system or even total derivatives that account for the component solution; only partial derivatives of residuals are needed.

SAND is an intrusive approach because it requires access to the residuals. These might not be available if components are provided as black boxes. Rather than computing coupling variables  $y_i$  and state variables  $u_i$  by converging the residuals to zero each component  $i$  just compute the current residuals  $\mathcal{R}_i$  for the current values of the coupling variables  $y$ , and the component states  $u_i$ .

#### 13.5.4 Modular Analysis and Unified Derivatives

The modular analysis and unified derivatives (MAUD) architecture is essentially MDF with built-in solvers and derivative computation that use the residual representation introduced in Section 13.3.2. <sup>37</sup> There are two main ideas in MAUD: 1) represent the coupled system as a

**Figure 13.13:** The SAND architecture lets the optimizer solve for all variables (design, coupling, and state variables) and component solvers are no longer needed.

<sup>37</sup>The MAUD architecture was developed by Hwang *et al.*<sup>37</sup>, who realized that the UDE provided the mathematical basis for a new MDO framework that makes sophisticated parallel solvers and coupled derivative computations available through a small set of user-defined functions.

<sup>37</sup>Hwang *et al.*, *A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives*. 2018

single nonlinear system and 2) linearize the coupled system using the UDE (6.74) and solve it for the coupled derivatives.

To represent the coupled system as a single nonlinear system, we view the MDA as a series of residuals and variables,  $R_i(u) = 0$ , corresponding to each component  $i = 1, \dots, n$ , as previously written in Eq. 13.1. Unlike the previous architectures, there is no distinction between the coupling variables and state variables; they are all just states,  $u$ . As previously shown in Fig. 13.5, the coupling variables can be considered to be the states by defining explicit components that translate the inputs and outputs.

In addition, both the design variables and functions of interest (objective and constraints) are also concatenated in the state variable vector. Denoting the original states for the coupled system (13.1) as  $\bar{u}$ , the new state is,

$$u \triangleq \begin{bmatrix} x \\ \bar{u} \\ f \end{bmatrix}. \quad (13.24)$$

We also need to augment the residuals to have a solvable system. The residuals corresponding to the design variables and output functions are formulated using the residual for explicit functions introduced in Eq. 13.2. The complete set of residuals is then,

$$R(u) \triangleq \begin{bmatrix} x - x_0 \\ r - R_{\bar{u}}(x, \bar{u}) \\ f - F(x, \bar{u}) \end{bmatrix}, \quad (13.25)$$

where  $x_0$  are fixed inputs, and  $F(x, \bar{u})$  is the actual computed value of the function. Formulating fixed inputs and explicit functions as residuals in this way might seem unnecessarily complicated, but it facilitates the formulation of the MAUD architecture, just like it did for the formulation of the UDE.

The two main ideas in MAUD mentioned above are directly associated with two main tasks: 1) the solution of the coupled system and 2) the computation of the coupled derivatives. The formulation of the concatenated states (13.24) and residuals (13.25) simplifies the implementation of the algorithms that perform the above tasks. To perform these tasks, MAUD assembles and solves four types of systems:

1. Fundamental system: This represents the numerical model and is in general a discretized nonlinear system of equations.

$$R(u) = 0$$

2. Newton step: A linear system based on the numerical model above whose solution yields an iteration toward the solution.

$$\frac{\partial R}{\partial u} \Delta u = -r$$

3. Forward differentiation (left equality of UDE): A linear system whose solution yields the derivative of all states with respect to one selected state. The state is selected by the appropriate choice of the column of the identity matrix. The selected states are usually the ones associated with  $x$ .

$$\frac{\partial R}{\partial u} \frac{du}{dr} = \mathcal{I}$$

4. Reverse differentiation (right equality of UDE): A linear system whose solution yields the derivative of one selected state with respect to all states. The state is selected by the appropriate choice of the column of the identity matrix. The selected states are usually the one associated with  $f$ .

$$\frac{\partial R^T}{\partial u} \frac{du^T}{dr} = \mathcal{I}$$

To efficiently solve the above systems of equations, MAUD provides the option for grouping the components of the fundamental system hierarchically. We show several examples of this grouping in Fig. 13.14. Of the two-component system on the left column, the top one has independent components that can be solved in parallel, while in the bottom one the components are coupled and need a coupled solver. The other systems have four components with different types dependencies that can be solved using two levels: the first level consists of two groups with two components each, and the higher level solves the two groups.

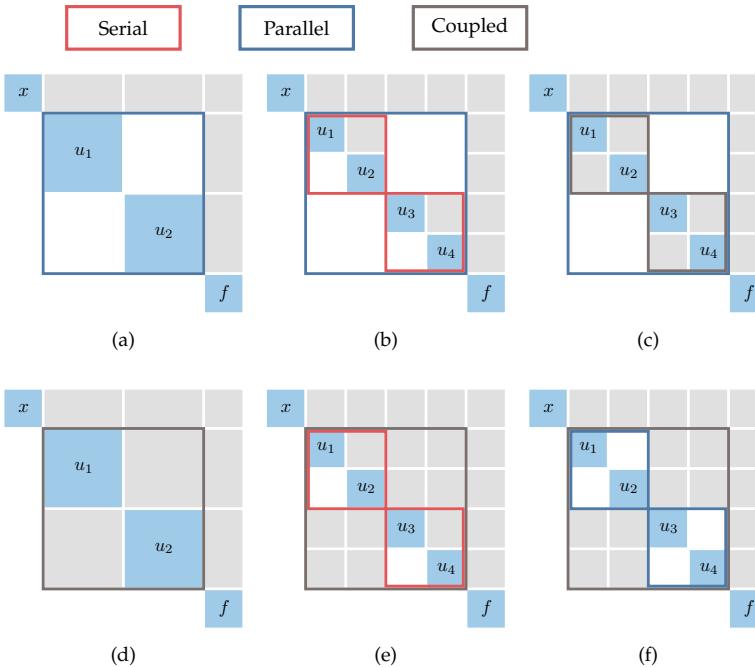
\*\*

## 13.6 Distributed Architectures

The monolithic MDO architectures we have covered so far form and solve a single optimization problem. Distributed architectures decompose this single optimization problem into a set of smaller optimization problems, or *disciplinary subproblems*, which are then coordinated by a *system-level subproblem*. One key requirement for these architectures is that they must be mathematically equivalent to the original monolithic problem so that they converge to the same solution.

<sup>\*\*</sup>MAUD was implemented in OpenMDAO V2 by Gray *et al.*<sup>89</sup>. OpenMDAO is an open-source framework developed by NASA to facilitate the development of multidisciplinary solvers. It includes all the features of MAUD introduced in this chapter and adds many other features, such as methods that take advantage of sparsity in the system coupling.

<sup>89</sup> Gray *et al.*, *OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization*. 2019



**Figure 13.14:** Example problem structures and corresponding MAUD hierarchical decompositions. The problem structure is shown using DSM. The hierarchical decompositions are shown above the matrices, where the components are the solid blue squares and the groups are the boxes (serial groups in red, parallel groups in blue, and coupled groups in gray).

There are two main motivations for distributed architectures. The first one is the possibility of decomposing the problem to reduce the computational time. The second motivation is to mimic the structure of large engineering design teams, where disciplinary groups have the autonomy to design their subsystem, so that MDO is more readily adopted in industry. Overall, distributed MDO architectures have fallen short on both of these expectations. Unless a problem has a special structure, there is no distributed architecture that converges as rapidly as a monolithic one. In practice, distributed architectures have not been used much recently.

There are two main types of distributed architectures: those that enforce multidisciplinary feasibility via an MDA somewhere in the process and those that enforce multidisciplinary feasibility in some other way (using constraints or penalties at the system level). This is analogous to MDF and IDF, respectively, so we name these types “distributed MDF” and “distributed IDF”.

### 13.6.1 Sequential Optimization

The sequential optimization approach is not considered to be an MDO architecture because in general, it does not converge to the optimum of the MDO problem. However, this is an intuitive approach to distributing the optimization of a system with multiple coupled components. This

approach does not include a system-level subproblem. Instead, each component is optimized in turn with respect to its local variable while satisfying its constraints. This is an approach that is often used in industry, where engineers are grouped by discipline, physical subsystem, or both. This makes sense when the engineering system being designed is too complex and the number of engineers too large to coordinate a simultaneous design involving all groups.

The sequential optimization approach is analogous to a block-Gauss-Seidel iteration, but in addition to solving for the component state variables we also solve an optimization problem for the design variables of that component. We can also view this approach as coordinate descent, except that instead of optimizing one variable at the time, we optimize a set of variables at the time.

### 13.6.2 Collaborative Optimization

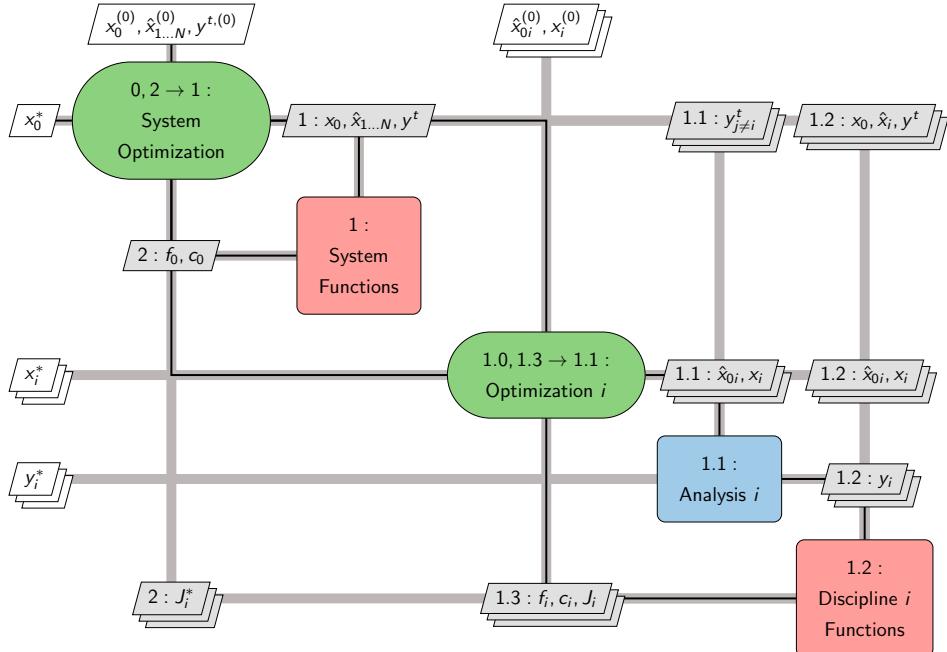
The collaborative optimization (CO) MDO architecture is inspired on how disciplinary teams work in the design of complex engineered systems. This is a distributed IDF architecture, where the disciplinary optimization problems are formulated to be independent of each other by using target values of the coupling and global design variables. These target values are then shared with all disciplines during every iteration of the solution procedure. The complete independence of disciplinary subproblems combined with the simplicity of the data-sharing protocol makes this architecture attractive for problems with a small amount of shared data.

The XDSM for CO is shown in Fig. 13.15. The system-level subproblem is similar to the original optimization problem except that: (1) local constraints are removed, (2) target coupling variables ( $y^t$ ) are added as design variables, and (3) a *consistency constraint* that quantifies the difference between the target coupling variables and actual coupling variables is added. This optimization problem can be written as

$$\begin{aligned}
 & \text{minimize} && f_0(x_0, \hat{x}_1, \dots, \hat{x}_N, y^t) \\
 & \text{by varying} && x_0, \hat{x}_1, \dots, \hat{x}_N, y^t \\
 & \text{subject to} && c_0(x_0, \hat{x}_1, \dots, \hat{x}_N, y^t) \leq 0 \\
 & && J_i^* = \|\hat{x}_{0i} - x_0\|_2^2 + \|\hat{x}_i - x_i\|_2^2 + \\
 & && \|y_i^t - y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t)\|_2^2 = 0 \quad \text{for } i = 1, \dots, N
 \end{aligned} \tag{13.26}$$

where  $\hat{x}_{0i}$  are copies of the global design variables that passed to discipline  $i$  and  $\hat{x}_i$  are copies of the local design variables passed to

the system subproblem. The constraint function  $J_i^*$  is a measure of the inconsistency between the values requested by the system-level subproblem and the results from the discipline  $i$  subproblem.



For each system-level iteration, the disciplinary subproblems do not include the original objective function. Instead the objective of each subproblem is to minimize the inconsistency function. For each discipline  $i$  the subproblem is

$$\begin{aligned} & \text{minimize} \quad J_i \left( \hat{x}_{0i}, x_i, y_i \left( \hat{x}_{0i}, x_i, y_{j\neq i}^t \right) \right) \\ & \text{by varying} \quad \hat{x}_{0i}, x_i \\ & \text{subject to} \quad c_i \left( \hat{x}_{0i}, x_i, y_i \left( \hat{x}_{0i}, x_i, y_{j\neq i}^t \right) \right) \leq 0. \end{aligned} \quad (13.27)$$

These subproblems are independent of each other and can be solved in parallel. Thus, the system-level subproblem is responsible for minimizing the design objective, while the discipline subproblems minimize system inconsistency while satisfying local constraints. The CO procedure is detailed in Alg. 13.7.<sup>††</sup> They formulated two versions of the CO architecture: CO<sub>1</sub> and CO<sub>2</sub>. The version presented in

Figure 13.15: Diagram for the CO architecture.

<sup>††</sup>Braun<sup>144</sup> showed that the CO problem statement is mathematically equivalent to the original MDO problem.

<sup>144</sup>A. Braun, *Collaborative Optimization: An Architecture for Large-Scale Distributed Design*. 1996

Section 13.6.2 is CO<sub>2</sub>.

---

**Algorithm 13.7: Collaborative optimization**


---

**Inputs:**

$x$ : *Initial design variables*

**Outputs:**

$x^*$ : *Optimal variables*

$f^*$ : *Corresponding objective value*

$c^*$ : *Corresponding constraint values*

---

0: Initiate system optimization iteration

**repeat**

    1: Compute system subproblem objectives and constraints

**for** Each discipline  $i$  (in parallel) **do**

            1.0: Initiate disciplinary subproblem optimization

**repeat**

                1.1: Evaluate disciplinary analysis

                1.2: Compute disciplinary subproblem objective and constraints

                1.3: Compute new disciplinary subproblem design point and  $J_i$

**until** 1.3 → 1.1: Optimization  $i$  has converged

**end for**

    2: Compute a new system subproblem design point

**until** 2 → 1: System optimization has converged

---

In spite of the organizational advantage of having fully separate disciplinary subproblems, CO suffers from numerical ill-conditioning. This is because the constraint gradients of the system problem at an optimal solution are all zero vectors, which violates the constraint qualification requirement for the KKT conditions. This slows down convergence when using a gradient-based optimization algorithm or prevents convergence all together.

### 13.6.3 Analytical Target Cascading

Analytical target cascading (ATC) is a distributed IDF architecture that uses penalties in the objective function to minimize the difference between the target variables requested by the system-level optimization and the actual variables computed by each discipline. This is an idea similar to the CO architecture in the previous section, except that ATC

uses penalties instead of a constraint. The ATC system-level problem is

$$\begin{aligned} \text{minimize } & f_0(x, y^t) + \sum_{i=1}^N \Phi_i(\hat{x}_{0i} - x_0, y_i^t - y_i(x_0, x_i, y^t)) + \\ & \Phi_0(c_0(x, y^t)) \end{aligned} \quad (13.28)$$

by varying  $x_0, y^t$ ,

where  $\Phi_0$  is a penalty relaxation of the global design constraints and  $\Phi_i$  is a penalty relaxation of the discipline  $i$  consistency constraints. The  $i^{th}$  discipline subproblem is:

$$\begin{aligned} \text{minimize } & f_0(\hat{x}_{0i}, x_i, y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t), y_{j \neq i}^t) + f_i(\hat{x}_{0i}, x_i, y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t)) + \\ & \Phi_i(y_i^t - y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t), \hat{x}_{0i} - x_0) + \\ & \Phi_0(c_0(\hat{x}_{0i}, x_i, y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t), y_{j \neq i}^t)) \\ \text{by varying } & \hat{x}_{0i}, x_i \\ \text{subject to } & c_i(\hat{x}_{0i}, x_i, y_i(\hat{x}_{0i}, x_i, y_{j \neq i}^t)) \leq 0. \end{aligned} \quad (13.29)$$

While the most common penalty functions in ATC are quadratic penalty functions, other penalty functions are possible. As mentioned in Section 5.3, penalty methods require a good selection of the penalty weight values to converge fast and accurately enough.

Fig. 13.16 shows the ATC architecture XDSM, where  $w$  denotes the penalty function weights used in the determination of  $\Phi_0$  and  $\Phi_i$ . The details of ATC are described in Alg. 13.8.

---

**Algorithm 13.8: Analytical target cascading**

---

**Inputs:**

$x$ : *Initial design variables*

**Outputs:**

$x^*$ : *Optimal variables*

$f^*$ : *Corresponding objective value*

$c^*$ : *Corresponding constraint values*

---

0: Initiate main ATC iteration

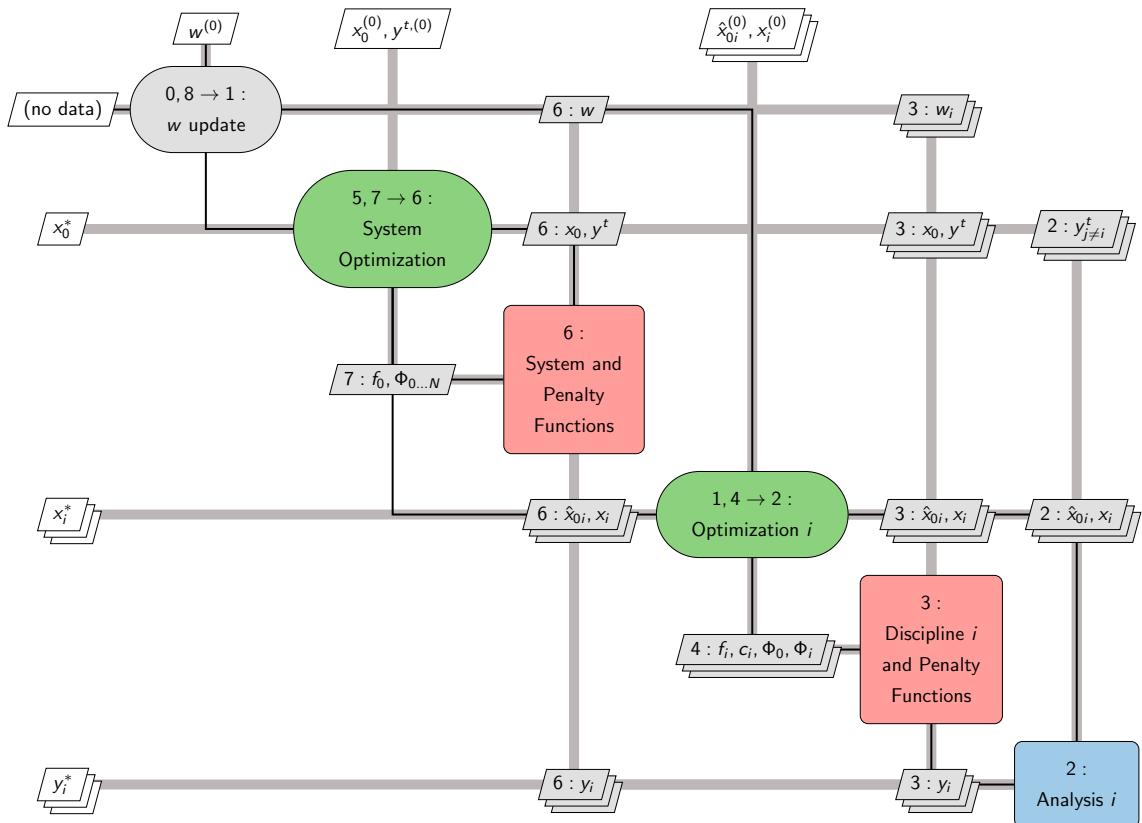
**repeat**

**for** Each discipline  $i$  **do**

    1: Initiate discipline optimizer

**repeat**

      2: Evaluate disciplinary analysis



3: Compute discipline objective and constraint functions and  
penalty function values

4: Update discipline design variables

**until**  $4 \rightarrow 2$ : Discipline optimization has converged

**end for**

5: Initiate system optimizer

**repeat**

6: Compute system objective, constraints, and all penalty functions

7: Update system design variables and coupling targets.

**until**  $7 \rightarrow 6$ : System optimization has converged

8: Update penalty weights

**until**  $8 \rightarrow 1$ : Penalty weights are large enough

**Figure 13.16:** Diagram for the ATC architecture

### 13.6.4 Bilevel Integrated System Synthesis

Bilevel integrated system synthesis (BLISS) uses a series of linear approximations to the original design problem, with bounds on the design variable steps, to prevent the design point from moving so far away that the approximations are too inaccurate. This is an idea similar to that of trust-region methods in Section 4.5. These approximations are constructed at each iteration using coupled derivatives (see Section 13.4). The system level subproblem is formulated as

$$\text{minimize } (f_0^*)_0 + \left( \frac{df_0^*}{dx_0} \right) \Delta x_0$$

by varying  $\Delta x_0$

$$\begin{aligned} \text{subject to } & (c_0^*)_0 + \left( \frac{dc_0^*}{dx_0} \right) \Delta x_0 \leq 0 \\ & (c_i^*)_0 + \left( \frac{dc_i^*}{dx_0} \right) \Delta x_0 \leq 0 \quad \text{for } i = 1, \dots, N \\ & \Delta x_{0L} \leq \Delta x_0 \leq \Delta x_{0U}. \end{aligned} \tag{13.30}$$

The discipline  $i$  subproblem is given by

$$\begin{aligned} \text{minimize } & (f_0)_0 + \left( \frac{df_0}{dx_i} \right) \Delta x_i \\ \text{by varying } & \Delta x_i \\ \text{subject to } & (c_0)_0 + \left( \frac{dc_0}{dx_i} \right) \Delta x_i \leq 0 \\ & (c_i)_0 + \left( \frac{dc_i}{dx_i} \right) \Delta x_i \leq 0 \\ & \Delta x_{iL} \leq \Delta x_i \leq \Delta x_{iU}. \end{aligned} \tag{13.31}$$

The extra set of constraints in both system-level and discipline subproblems denote the design variables bounds. To prevent violation of the disciplinary constraints by changes in the shared design variables, post-optimality derivatives (the change in the optimized disciplinary constraints with respect to a change in the system design variables) are required to solve the system-level subproblem.

---

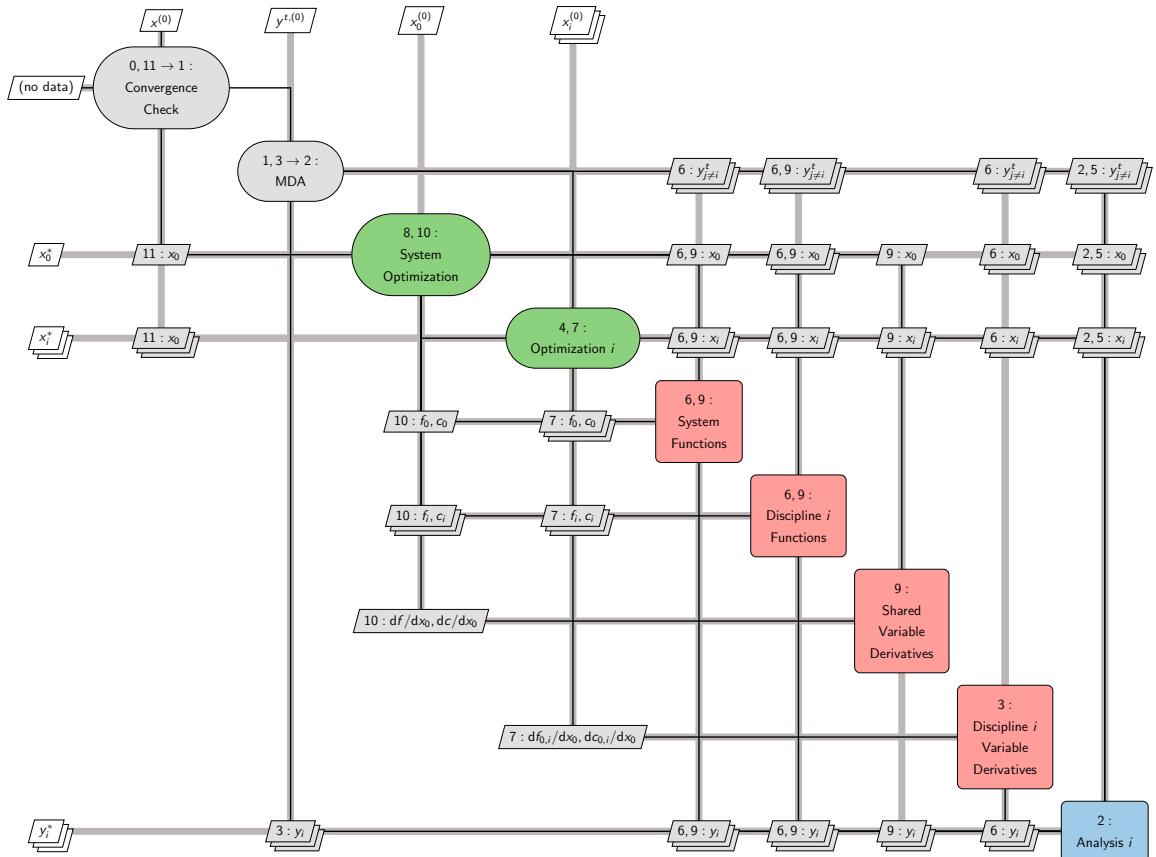
**Algorithm 13.9:** Bilevel integrated system synthesis

**Inputs:**

$x$ : *Initial design variables*

**Outputs:**

$x^*$ : *Optimal variables*



$f^*$ : Corresponding objective value

$c^*$ : Corresponding constraint values

Figure 13.17: Diagram for the BLISS architecture

---

```

0: Initiate system optimization
repeat
  1: Initiate MDA
  repeat
    2: Evaluate discipline analyses
    3: Update coupling variables
  until 3 → 2: MDA has converged
  4: Initiate parallel discipline optimizations
  for Each discipline  $i$  do
    5: Evaluate discipline analysis
    6: Compute objective and constraint function values and derivatives
    with respect to local design variables
    7: Compute the optimal solutions for the disciplinary subproblem
  end for

```

- 
- 8: Initiate system optimization
  - 9: Compute objective and constraint function values and derivatives with respect to shared design variables using post-optimality analysis
  - 10: Compute optimal solution to system subproblem
  - until** 11 → 1: System optimization has converged
- 

Figure 13.17 shows the XDSM for BLISS and the corresponding steps are listed in Alg. 13.9. Since BLISS uses an MDA, it is a distributed MDF architecture. Due to the linear nature of the optimization problems, repeated interrogation of the objective and constraint functions is not necessary once we have the gradients. If the underlying problem is highly nonlinear, the algorithm may converge slowly. The variable bounds may help the convergence if these bounds are properly chosen, such as through a trust region framework.

### 13.6.5 Asymmetric Subspace Optimization

Asymmetric subspace optimization (ASO) is a distributed MDF architecture that is motivated by cases where there is a large discrepancy between the cost of the disciplinary solvers. To reduce the number of the more expensive disciplinary analysis, the cheaper disciplinary analyses are replaced by disciplinary design optimizations inside the overall MDA.

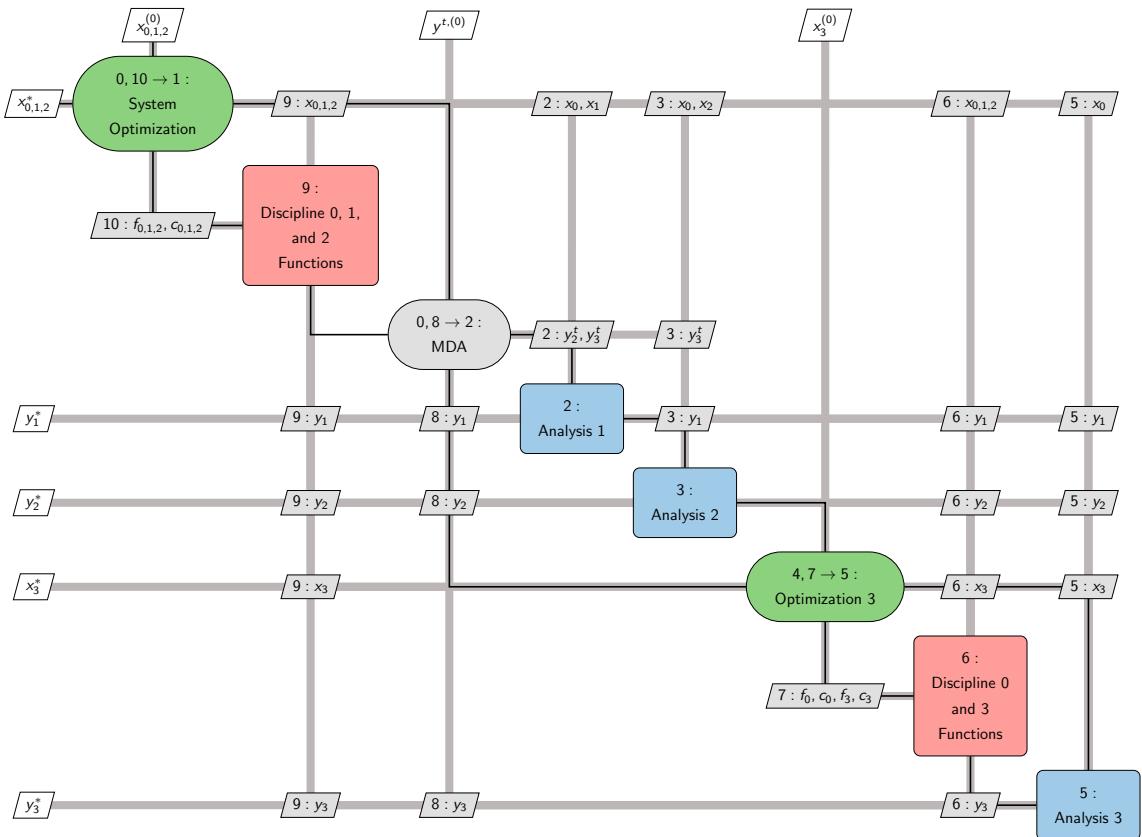
The system-level optimization subproblem is

$$\begin{aligned}
 & \text{minimize} \quad f_0(x, y(x, y)) + \sum_k f_k(x_0, x_k, y_k(x_0, x_k, y_{j \neq k})) \\
 & \text{by varying} \quad x_0, x_k \\
 & \text{subject to} \quad c_0(x, y(x, y)) \leq 0 \\
 & \quad c_k(x_0, x_k, y_k(x_0, x_k, y_{j \neq k})) \leq 0 \quad \text{for all } k,
 \end{aligned} \tag{13.32}$$

where subscript  $k$  denotes disciplinary information that remains outside of the MDA. The disciplinary optimization subproblem for discipline  $i$ , which is resolved inside the MDA, is

$$\begin{aligned}
 & \text{minimize} \quad f_0(x, y(x, y)) + f_i(x_0, x_i, y_i(x_0, x_i, y_{j \neq i})) \\
 & \text{by varying} \quad x_i \\
 & \text{subject to} \quad c_i(x_0, x_i, y_i(x_0, x_i, y_{j \neq i})) \leq 0.
 \end{aligned} \tag{13.33}$$

Figure 13.18 shows a three-discipline case where the third discipline is replaced with a design optimization. The corresponding sequence of operations in ASO is listed in Alg. 13.10.



Algorithm 13.10: ASO

**Inputs:** $x$ : Initial design variables**Outputs:** $x^*$ : Optimal variables $f^*$ : Corresponding objective value $c^*$ : Corresponding constraint values

0: Initiate system optimization

**repeat**

1: Initiate MDA

**repeat**

2: Evaluate Analysis 1

3: Evaluate Analysis 2

4: Initiate optimization of Discipline 3

**repeat**

5: Evaluate Analysis 3

Figure 13.18: Diagram for the ASO architecture

---

```

6: Compute discipline 3 objectives and constraints
7: Update local design variables
until 7 → 5: Discipline 3 optimization has converged
8: Update coupling variables
until 8 → 2 MDA has converged
9: Compute objective and constraint function values for all disciplines 1
and 2
10: Update design variables
until 10 → 1: System optimization has converged

```

---

For a gradient-based system-level optimizer, the gradients of the objective and constraints must take into account the suboptimization. This requires coupled post-optimality derivative computation, which increases the cost of both computational and implementation time compared a normal coupled derivative computation. The total optimization cost is only competitive with MDF if the discrepancy between each disciplinary solver is high enough.

### 13.6.6 Other Distributed Architectures

There are other distributed MDF architectures other than BLISS and ASO: concurrent subspace optimization (CSSO) and MDO of independent subspaces (MDOIS)

CSSO requires surrogate models for the analyses for all disciplines. The system-level optimization subproblem is solved based on the surrogate models and is therefore fast. The discipline-level optimization subproblem uses the actual analysis from the corresponding discipline and surrogate models for all other disciplines. The solutions for each discipline subproblem is used to update the surrogate models.

MDOIS only applies when no global variables exist. In this case, discipline subproblems are solved independently assuming fixed coupling variables, and then an MDA is performed to update the coupling.

There are also other distributed IDF architectures. Some of these are like CO in that they use a multilevel approach to enforce multidisciplinary feasibility: BLISS-2000 and quasi-separable decomposition (QSD). Other architectures enforce multidisciplinary feasibility with penalties, like ATC: inexact penalty decomposition (IPD), exact penalty decomposition (EPD), and enhanced collaborative optimization (ECO).

BLISS-2000 is a variation of BLISS that uses surrogate models to represent the coupling variables for all disciplines. Each discipline subproblem minimizes the linearized objective with respect to local variables subject to local constraints. The system-level subproblem

minimizes the objective with respect to the global variables and coupling variables while enforcing consistency constraints.

When using QSD, the objective and constraint functions are assumed to be dependent only on the shared design variables and coupling variables. Each discipline is assigned a “budget” for a local objective and the discipline problems maximize the margin in their local constraints and the budgeted objective. The system-level subproblem minimizes the objective and budgets of each discipline while enforcing the global constraints and a positive margin for each discipline.

IPD and EPD are applicable to MDO problems with no global objectives or constraints. They are similar to ATC in that copies of the share variables are used for every discipline subproblem and the consistency constraints are relaxed with a penalty function. Unlike ATC, however, the simpler structure of the discipline subproblems is exploited to compute post-optimality derivatives to guide the system-level optimization subproblem.

Like CO, ECO uses copies of the global variables. The discipline subproblems minimize quadratic approximations of the objective while enforcing local constraints and linear models of the nonlocal constraints. The system-level subproblem minimizes the total violation of all consistency constraints with respect to the global variables.

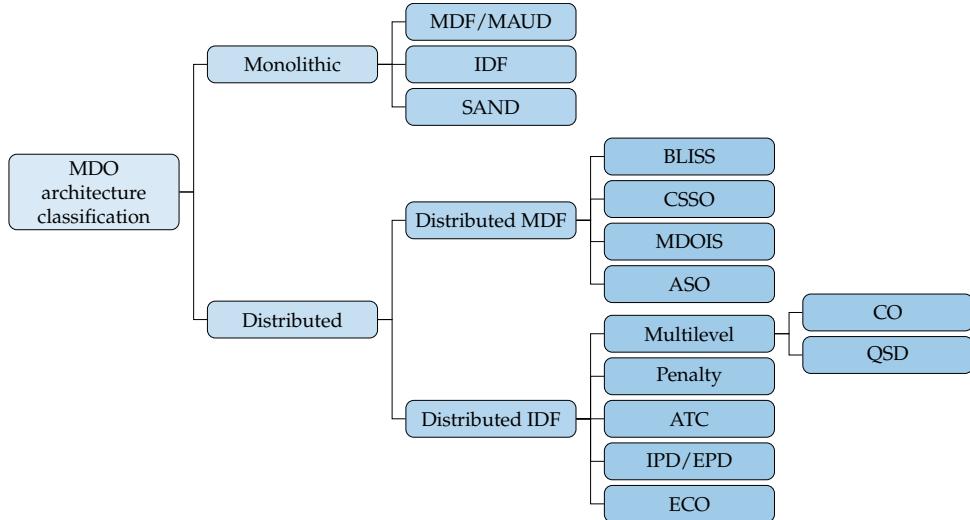
## 13.7 Summary

MDO architectures provide different options for solving MDO problems. An acceptable MDO architecture must be mathematically equivalent to the original problem and thus converge to the same optima. Sequential optimization, while intuitive, is not mathematically equivalent to the original problem and yields a design inferior to the MDO optimum.

MDO architectures are divided into two broad categories, as shown in Fig. 13.19: monolithic architecture and distributed architectures. Monolithic architectures solve a single optimization problem, while distributed architecture solve optimization subproblems for each discipline and a system-level optimization problem. Overall, monolithic architectures exhibit a much better convergence rate than distributed architectures.<sup>145</sup> In the last few years, the vast majority of MDO applications have used monolithic MDO architectures.

The distributed architectures can be divided according to whether they enforce multidisciplinary feasibility (through an MDA of the whole system), or not. Distributed MDF architectures enforce multidisciplinary feasibility through an MDA. The distributed IDF architectures are like IDF in that no MDA is required. However, they must ensure

<sup>145</sup> Tedford *et al.*, *Benchmarking Multidisciplinary Design Optimization Algorithms*. 2010



multidisciplinary feasibility in some other way. Some do this by formulating an appropriate multilevel optimization (such as CO) and others use penalties to ensure this (such as ATC). <sup>#</sup>

There are a number of commercial MDO frameworks that are available, including Isight/SEE <sup>146</sup> by Dassault Systèmes, ModelCenter/-CenterLink by Phoenix Integration, modeFRONTIER by Esteco, AML Suite by TechnoSoft, Optimus by Noesis Solutions, and VisualDOC by TechnoSoft's AML suite, Noesis Solutions' Optimus, and VisualDOC by Vanderplaats Research and Development <sup>147</sup>. These frameworks focus on making it easy for users to couple multiple disciplines and to use the optimization algorithms through graphical user interfaces. They also provide convenient wrappers to popular commercial engineering tools. While this focus has made it convenient for users to implement and solve MDO problems, the numerical methods used to converge the multidisciplinary analysis (MDA) and the optimization problem are usually not as sophisticated as the methods presented in this book. For example, these frameworks often use fixed-point iteration to converge the MDA. When derivatives are needed for a gradient-based optimizer, finite-difference approximations are used rather than more accurate analytic derivatives.

**Figure 13.19:** Classification of MDO architectures.

<sup>#</sup>Martins *et al.*<sup>34</sup> describes all these MDO architectures in detail.

<sup>34</sup>. Martins *et al.*, *Multidisciplinary Design Optimization: A Survey of Architectures*. 2013

<sup>146</sup>. Golovidov *et al.*, *Flexible implementation of approximation concepts in an MDO framework*. 1998

<sup>147</sup>. Balabanov *et al.*, *VisualDOC: A Software System for General Purpose Integration and Design Optimization*. 2002

## Problems

13.1 Answer *true* or *false* and justify your answer.

- We prefer to use the term “component” instead of “discipline” because it is more general.

- b) Local design variables affect only one discipline in the MDO problem, while global variables affect all disciplines.
- c) All multidisciplinary models can be written in the functional form, but not all can be written in the residual form.
- d) The coupling variables are a subset of component state variables.
- e) Multidisciplinary models can be represented by directed cyclic graphs where the nodes represent components and edges represent coupling variables.
- f) The nonlinear block Jacobi and Gauss–Seidel methods can be used with any combination of component solvers.
- g) All the derivative computation methods from Chapter 6 can be implemented for coupled multidisciplinary systems.
- h) Implicit analytic methods for derivative computation are incompatible with the functional form of multidisciplinary models.
- i) The modular analysis and unified derivatives architecture is based on the unified derivatives equation.
- j) The MDF architecture has fewer design variables and more constraints than IDF.
- k) The main difference between monolithic and distributed MDO architectures is that the distributed architectures perform optimization at multiple levels.
- l) Sequential optimization is a valid MDO approach but the main disadvantage is that it converges slowly.

13.2 Pick a multidisciplinary engineering system from the literature or formulate one based on your experience.

- a) Identify the different analyses and coupling variables.
- b) List the design variables and classify them as local or global.
- c) Identify the objective and constraint functions.
- d) Draw a diagram similar to the one in Fig. 13.1 for your system.
- e) *Exploration:* Think about the objective that each discipline would have if considered separately and discuss the trades needed to optimize the multidisciplinary objective.

## Mathematics Review

# A

This chapter briefly reviews select mathematical concepts that are used throughout the book.

By the end of this chapter you should be able to:

1. Compute and understand the difference between partial and total derivatives
2. Identify and use vector norms.
3. Perform matrix multiplications and compute derivatives of matrix functions.
4. Perform Taylor's series expansions.

### A.1 Chain Rule, Partial Derivatives, and Total Derivatives

The single variable chain rule is needed for differentiating composite functions. Given a composite function,  $f(g(x))$ , the derivative with respect to the variable  $x$  is:

$$\frac{d}{dx}(f(g(x))) = \frac{df}{dg} \frac{dg}{dx} \quad (\text{A.1})$$

---

Example A.1: Single variable chain rule.

Let  $f(g(x)) = \sin(x^2)$ . In this case,  $f(g) = \sin(g)$ , and  $g(x) = x^2$ . The derivative with respect to  $x$  is:

$$\frac{d}{dx}(f(g(x))) = \frac{d}{dg}(\sin(g)) \frac{d}{dx}(x^2) = \cos(x^2)(2x) \quad (\text{A.2})$$

---

If a function depends on more than one variable, then we need to distinguish between *partial* and *total* derivatives. For example, if

$f(g(x), h(x))$  then  $f$  is a function of two variables:  $g$  and  $h$ . The application of the chain rule for this function is:

$$\frac{d}{dx}(f(g(x), h(x))) = \frac{\partial f}{\partial g} \frac{dg}{dx} + \frac{\partial f}{\partial h} \frac{dh}{dx} \quad (\text{A.3})$$

where  $\partial/\partial x$  indicates a partial derivative and  $d/dx$  is a total derivative. When taking a partial derivative, we take the derivative with respect to only that variable, treating all other variables as constants. More generally,

$$\frac{d}{dx}(f(g_1(x), \dots, g_n(x))) = \sum_{i=1}^n \left( \frac{\partial f}{\partial g_i} \frac{dg_i}{dx} \right) \quad (\text{A.4})$$

---

**Example A.2:** Partial versus total derivatives.

Consider  $f(x, y(x)) = x^2 + y^2$  where  $y(x) = \sin(x)$ . The *partial* derivative of  $f$  with respect to  $x$  is:

$$\frac{\partial f}{\partial x} = 2x \quad (\text{A.5})$$

whereas the *total* derivative of  $f$  with respect to  $x$  is:

$$\begin{aligned} \frac{df}{dx} &= \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{dy}{dx} \\ &= 2x + 2y \cos(x) \\ &= 2x + 2 \sin(x) \cos(x) \end{aligned} \quad (\text{A.6})$$

Notice that the partial derivative and total derivative are quite different. For this simple case we could also find the total derivative by direct substitution and then using an ordinary one-dimensional derivative. Substituting in  $y(x) = \sin(x)$  directly into the original expression for  $f$ :

$$f(x) = x^2 + \sin^2(x) \quad (\text{A.7})$$

$$\frac{df}{dx} = 2x + 2 \sin(x) \cos(x) \quad (\text{A.8})$$


---

**Example A.3:** Multivariable chain rule.

Expanding on our single variable example, let  $g(x) = \cos(x)$  and  $h(x) = \sin(x)$  and  $f(g, h) = g^2 h^3$ . Then  $f(g(x), h(x)) = \cos^2(x) \sin^3(x)$ . Applying Eq. A.3 we have:

$$\begin{aligned} \frac{d}{dx}(f(g(x), h(x))) &= \frac{\partial f}{\partial g} \frac{dg}{dx} + \frac{\partial f}{\partial h} \frac{dh}{dx} \\ &= 2gh^3 \frac{dg}{dx} + g^2 3h^2 \frac{dh}{dx} \\ &= -2gh^3 \sin(x) + g^2 3h^2 \cos(x) \\ &= -2 \cos(x) \sin^4(x) + 3 \cos^3(x) \sin^2(x) \end{aligned} \quad (\text{A.9})$$

## A.2 Vector and Matrix Norms

The most familiar norm for vectors is the 2-norm, which corresponds to the Euclidean length of the vector:

$$\|x\|_2 = (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2}, \quad (\text{A.10})$$

Because this norm is used so often, we often omit the subscript and just write  $\|x\|$ . More generally, we can refer to a class of norms called  $p$ -norms:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{1/p} \quad (\text{A.11})$$

Of all the  $p$ -norms, three that are most commonly used: the 2-norm we just discussed, the 1-norm, and the  $\infty$ -norm. From the above definition, we see that the 1-norm is the sum of the absolute values of all the entries in  $x$ .

$$\|x\|_1 = |x_1| + |x_2| + \dots + |x_n| \quad (\text{A.12})$$

The application of  $\infty$  in the  $p$ -norm definition is perhaps less obvious, but as  $p \rightarrow \infty$  the largest term in that sum dominates all of the others. Raising that quantity to the power of  $1/p$  causes the exponents to cancel, leaving only the largest magnitude component of  $x$ . This is exactly how the infinity norm is defined:

$$\|x\|_\infty = \max_i |x_i| \quad (\text{A.13})$$

The infinity norm is commonly used in connection with optimization convergence criteria.

Several norms for matrices exist as well. One that is used in this book is the Frobenius norm:

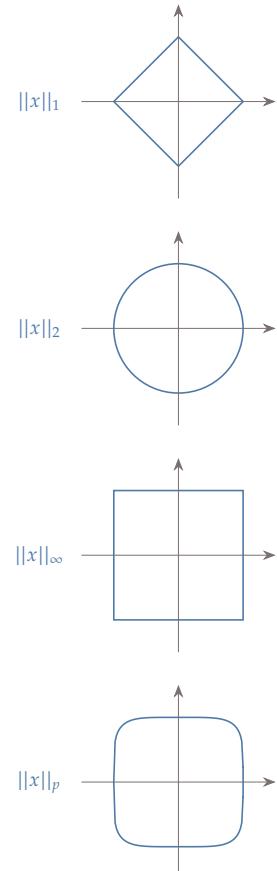
$$\|H\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n H_{ij}^2}, \quad (\text{A.14})$$

where  $H$  is an  $m \times n$  matrix.

## A.3 Matrix Multiplication

Consider a matrix  $A \in \mathbb{R}^{mxn*}$  and a matrix  $B \in \mathbb{R}^{nxp}$ . The two matrices can be multiplied together ( $C = AB$ ) as follows:

$$C_{ij} = \sum_{k=1}^n A_{ik}B_{kj} \quad (\text{A.15})$$



**Figure A.1:** Norms for two-dimensional case.

\*This means that the matrix is comprised of real numbers and that it has  $m$  rows and  $n$  columns.

where  $C \in \mathbb{R}^{m \times p}$ . Notice that two matrices can be multiplied only if their inner dimensions are equal ( $n$  in this case). The remaining products discussed in section are just special cases of matrix multiplication, but are common enough that we discuss them separately.

### A.3.1 Vector-Vector Products

In this book, a vector  $u$  is a column vector, thus the row vector would be represented as  $u^T$ . The product of two vectors can be performed in two ways. The more common is called an *inner product* (also known as a *dot product*, or scalar product). The inner product, is a functional, meaning it is an operator that acts on vectors and produces a scalar. In the real vector space,  $\mathbb{R}^n$ , the inner product of two vectors,  $u$  and  $v$ , whose dimension is equal, is defined algebraically as:

$$u^T v = [u_1 \ u_2 \ \dots \ u_n] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^n u_i v_i \quad (\text{A.16})$$

Notice that the order is irrelevant:

$$u^T v = v^T u \quad (\text{A.17})$$

In Euclidean space, where vectors possess magnitude and direction, the inner product is defined as

$$u^T v = \|u\| \|v\| \cos(\theta) \quad (\text{A.18})$$

where  $\|\cdot\|$  indicates a 2-norm, and  $\theta$  is the angle between the two vectors.

An alternative vector-vector product is the *outer product*, which takes the two vectors and multiplies them element-wise to produce a matrix. Note that the outer product, unlike the inner product, does not require the vectors to be of the same length.

$$uv^T = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} [v_1 \ v_2 \ \dots \ v_n] = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{bmatrix} \quad (\text{A.19})$$

or in index form:

$$(uv^T)_{ij} = u_i v_j \quad (\text{A.20})$$

### A.3.2 Matrix-Vector Products

Consider multiplying a matrix  $A \in \mathbb{R}^{m \times n}$  by a vector  $u \in \mathbb{R}^n$ . The result is a vector  $v \in \mathbb{R}^m$ .

$$v = Au \Rightarrow v_i = \sum_{j=1}^n A_{ij} u_j \quad (\text{A.21})$$

We can see that the entries in  $v$  are dot products between the rows of  $A$  and  $u$ :

$$v = \begin{bmatrix} | & a_1^T & | \\ | & a_2^T & | \\ \vdots & & \\ | & a_m^T & | \end{bmatrix} u \quad (\text{A.22})$$

where  $a_j^T$  is the  $j^{\text{th}}$  row of the matrix  $A$ .

Alternatively, it could be thought of as a linear combination of the columns of  $A$  where the  $u_j$  are the weights:

$$v = \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} u_1 + \begin{bmatrix} | \\ a_2 \\ | \end{bmatrix} u_2 + \dots + \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} u_n \quad (\text{A.23})$$

where  $a_i$  are the columns of  $A$ .

We can also multiply by a vector on the left, instead of on the right:

$$v^T = u^T A \quad (\text{A.24})$$

In this case a row vector is multiplied against a matrix producing a row vector.

### A.3.3 Quadratic Form (Vector-Matrix-Vector Product)

Another common product is a *quadratic form*. A quadratic form consists of a row vector, times a matrix, times a column vector, producing a scalar:

$$\alpha = u^T A u = [u_1 \ u_2 \ \dots \ u_n] \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \quad (\text{A.25})$$

or in index form:

$$\alpha = \sum_{i=1}^n \sum_{j=1}^n u_i A_{ij} u_j \quad (\text{A.26})$$

In general, a vector-matrix-vector product can have a non-square  $A$  matrix, and the vectors would be two different sizes, but for a quadratic

form the two vectors  $u$  are identical and thus  $A$  is square. Also in a quadratic form we assume that  $A$  is symmetric (even if it isn't, only the symmetric part of  $A$  contributes anyway so effectively it acts like a symmetric matrix).

#### A.4 Matrix Types

There are several common types of matrices that appear regularly throughout this book. We review some terminology here.

A *diagonal matrix* is a matrix where all off-diagonal terms are zero. In other words  $A$  is diagonal if:

$$A_{ij} = 0 \text{ for all } i \neq j \quad (\text{A.27})$$

The *identity matrix*  $I$  is a special diagonal matrix where all diagonal components are one.

The *transpose* of a matrix is defined as:

$$[A^T]_{ij} = A_{ji} \quad (\text{A.28})$$

Note that:

$$(A^T)^T = A \quad (\text{A.29})$$

$$(A + B)^T = A^T + B^T \quad (\text{A.30})$$

$$(AB)^T = B^T A^T \quad (\text{A.31})$$

A *symmetric matrix* is one where the matrix is equal to its transpose:

$$A_{ij} = A_{ji} \quad (\text{A.32})$$

The *inverse* of a matrix satisfies:

$$AA^{-1} = I = A^{-1}A \quad (\text{A.33})$$

Not all matrices are invertible. Some common properties for inverses are:

$$(A^{-1})^{-1} = A \quad (\text{A.34})$$

$$(AB)^{-1} = B^{-1}A^{-1} \quad (\text{A.35})$$

$$[A^{-1}]^T = [A^T]^{-1} \quad (\text{A.36})$$

A symmetric matrix  $A$  is *positive definite* if for all vectors  $\mathbf{x}$  in the real space:

$$\mathbf{x}^T A \mathbf{x} > 0 \quad (\text{A.37})$$

Similarly, a *positive semi-definite* matrix satisfies the condition:

$$x^T M x \geq 0 \quad (\text{A.38})$$

and a *negative definite* matrix satisfies the condition

$$x^T M x < 0 \quad (\text{A.39})$$

## A.5 Matrix Derivatives

Let's consider derivatives of a few common cases: linear and quadratic functions. Combining the concept of partial derivatives and matrix forms of equations allows us to find the gradients of matrix functions. First, let us look at a linear function,  $f$ , defined as

$$f(x) = a^T x + b = \sum_{i=1}^n a_i x_i + b_i \quad (\text{A.40})$$

where  $a$ ,  $x$ , and  $b$  are vectors of length  $n$ , and  $a_i$ ,  $x_i$ , and  $b_i$  are the  $i$ th elements of  $a$ ,  $x$ , and  $b$ , respectively. If we take the partial derivative of each element with respect to an arbitrary element of  $x$ , namely  $x_k$ , we get

$$\frac{\partial}{\partial x_k} \left[ \sum_{i=1}^n a_i x_i + b_i \right] = a_k \quad (\text{A.41})$$

Thus:

$$\nabla_x (a^T x + b) = a \quad (\text{A.42})$$

Recall the quadratic form presented in Appendix A.3.3, we can combine that with a linear term to form a general quadratic function:

$$f(x) = x^T A x + b^T x + c \quad (\text{A.43})$$

where  $x, b$  and  $c$  are still vectors of length  $n$ , and  $A$  is an  $n$  by  $n$  symmetric matrix. In index notation,  $f$  looks like

$$f(x) = \sum_{i=1}^n \sum_{j=1}^n x_i a_{ij} x_j + b_i x_i + c_i \quad (\text{A.44})$$

For convenience, we'll separate the diagonal terms from the off diagonal terms leaving us with

$$f(x) = \sum_{i=1}^n [a_{ii} x_i^2 + b_i x_i + c_i] + \sum_{j \neq i} x_i a_{ij} x_j \quad (\text{A.45})$$

Now we take the partial derivatives with respect to  $x_k$  as before yielding:

$$\frac{\partial f}{\partial x_k} = 2a_{kk}x_k + b_k + \sum_{j \neq i} x_j a_{jk} + \sum_{j \neq i} a_{kj}x_j \quad (\text{A.46})$$

We now move the diagonal terms back into the sums to get

$$\frac{\partial f}{\partial x_k} = b_k + \sum_{j=1}^n (x_j a_{jk} + a_{kj}x_j), \quad (\text{A.47})$$

which we can put back into matrix form as:

$$\nabla_x f(x) = A^T x + Ax + b \quad (\text{A.48})$$

If  $A$  is symmetric then  $A^T = A$  and thus:

$$\nabla_x(x^T A x + b^T x + c) = 2Ax + b \quad (\text{A.49})$$

## A.6 Taylor Series Expansion

Series expansions are representations of given function in terms of a series of other (usually simpler) functions. One common series expansion is the *Taylor series*, which is expressed as a polynomial whose coefficients are based on the derivatives of the original function at a fixed point.

The Taylor series is a general tool that can be applied whenever the function has derivatives. We can use this series to estimate the value of the function near the given point, which is useful when the function is difficult to evaluate directly. The Taylor series is used to derive algorithms for finding zeroes of functions and algorithms for minimizing functions.

To derive the Taylor series, we start with an infinite polynomial series about an arbitrary point,  $x$ , to approximate the value of a function at  $x + \Delta x$  using

$$f(x + \Delta x) = a_0 + a_1 \Delta x + a_2 \Delta x^2 + \dots + a_k \Delta x^k + \dots \quad (\text{A.50})$$

We can make this approximation exact at  $\Delta x = 0$  by setting the first coefficient to  $f(x)$ . To find the appropriate value for  $a_1$ , we take the first derivative to get

$$f'(x + \Delta x) = a_1 + 2a_2 \Delta x + \dots + ia_k \Delta x^{k-1} + \dots, \quad (\text{A.51})$$

which means that we need  $a_1 = f'(x)$  to obtain an exact derivative at  $x$ . To derive the other coefficients, we systematically take the derivative of

both sides and the appropriate value of the first nonzero term (which is always constant). Identifying the pattern yields the general formula for the  $n^{\text{th}}$ -order coefficient

$$a_k = \frac{f^{(k)}(x)}{k!}. \quad (\text{A.52})$$

Substituting this into the polynomial (A.50) yields the Taylor series

$$f(x + \Delta x) = \sum_{k=0}^{\infty} \frac{\Delta x^k}{k!} f^{(k)}(x). \quad (\text{A.53})$$

The series is typically truncated to use terms up to order  $m$ ,

$$f(x + \Delta x) = \sum_{k=0}^m \frac{\Delta x^k}{k!} f^{(k)}(x) + O(\Delta x^{m+1}), \quad (\text{A.54})$$

which yields an approximation with a truncation error of order  $O(\Delta x^{m+1})$ . In optimization, it is common use the first three terms (up to  $m = 2$ ) to get a quadratic approximation.

---

**Example A.4:** Taylor series expansion for single variable.

Consider the scalar function of a single variable,  $f(x) = x - 4 \cos(x)$ . If we use Taylor series expansions of this function about  $x = 0$ , we get

$$f(\Delta x) = -4 + \Delta x + 2\Delta x^2 - \frac{1}{6}\Delta x^4 + \frac{1}{180}\Delta x^6 - \dots \quad (\text{A.55})$$

Three different truncations of this series are plotted and compared to the exact function in Fig. A.2.

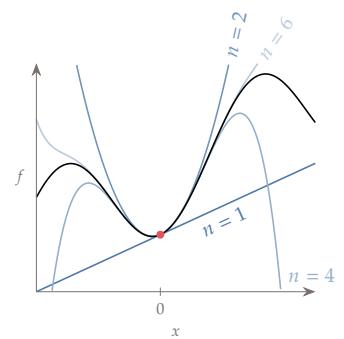
---

The Taylor series in multiple dimensions is similar to the single variable case, but more complicated. The first derivative of the function becomes a gradient vector and the second derivatives becomes a Hessian matrix. Also, we need to define a direction along which we want to approximate the function, since that information is not inherent as it is in a 1-D function. The Taylor series expansion in  $n$ -dimensions along a direction  $p$  can be written as

$$f(x + \alpha p) = f(x) + \alpha \sum_{k=1}^n p_k \frac{\partial f}{\partial x_k} + \frac{1}{2} \alpha^2 \sum_{k=1}^n \sum_{l=1}^n p_k p_l \frac{\partial^2 f}{\partial x_k \partial x_l} + O(\alpha^3), \quad (\text{A.56})$$

where  $\alpha$  is a scalar that determines how far to go in the direction  $p$ . In matrix form, we can write

$$f(x + \alpha p) = f(x) + \alpha \nabla f(x)^T p + \frac{1}{2} \alpha^2 p^T H(x) p + O(\alpha^3), \quad (\text{A.57})$$



**Figure A.2:** Taylor series expansions for 1-D example. The more terms we consider from the Taylor series, the better the approximation.

where  $H$  is the Hessian matrix.

---

**Example A.5:** Taylor series expansion for two variables.
 

---

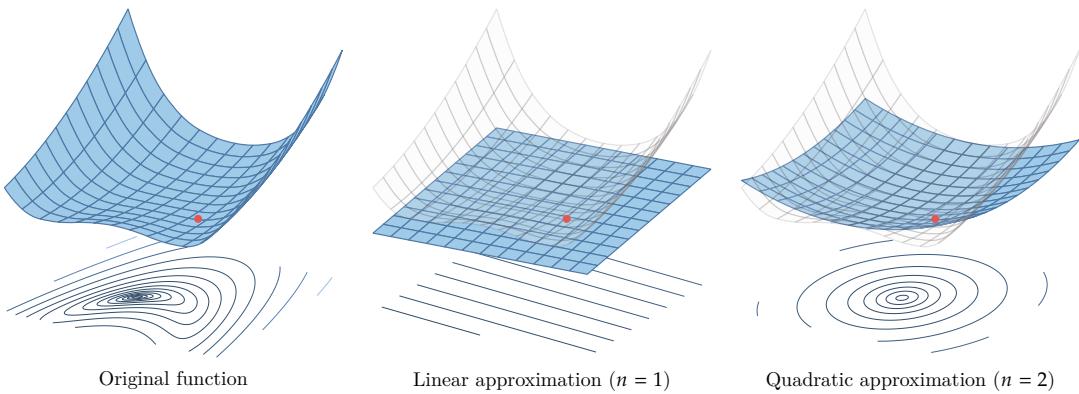
Consider the following function of two variables,

$$f(x_1, x_2) = (1 - x_1)^2 + (1 - x_2)^2 + \frac{1}{2} (2x_2 - x_1^2)^2. \quad (\text{A.58})$$

Performing a Taylor series expansion about  $x = [0, -2]^T$ , we get,

$$f(\alpha p) = 18 + \alpha [-2 \ -14] p + \frac{1}{2} \alpha^2 p^T \begin{bmatrix} 10 & 0 \\ 0 & 6 \end{bmatrix} p \quad (\text{A.59})$$

The original function, the linear approximation, and the quadratic approximation are compared in Fig. A.3.




---

**Figure A.3:** Taylor series approximations for 2-D example.

## Linear Solvers

# B

In Section 3.7 we present an overview of solution methods for discretized systems of equations, followed by an introduction to Newton-based methods for solving nonlinear equations. Here, we review solvers for linear systems, which are required to solve for each step of Newton-based methods.

By the end of this chapter you should be able to:

1. Understand the differences between direct and indirect methods for linear problems.
2. Understand how to apply these methods to nonlinear problems.

If the equations are linear, they can be written as

$$r(u) = b - Au = 0, \quad (\text{B.1})$$

where  $A$  is a square ( $n \times n$ ) matrix and  $b$  is a vector, and neither of these depend on  $u$ . To solve this linear system, we can use either a direct method or an iterative method.

### B.1 Direct Methods

The standard way to solve linear systems of equations in a computer is Gaussian elimination, which in matrix form is equivalent to *LU decomposition*. This is a decomposition (or factorization) of  $A$ , such as  $A = LU$ , where  $L$  is a unit lower triangular matrix, and  $U$  is an upper triangular matrix, as shown in Fig. B.1.

The decomposition transforms the matrix  $A$  into an upper-triangular matrix  $U$  by introducing zeros below the diagonal, one column at the time, starting with the first one and progressing from left to right. This is done by subtracting multiples of each row from subsequent rows. These operations can be expressed as sequence of multiplications with

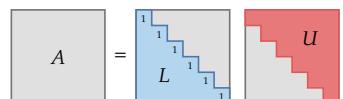


Figure B.1: LU decomposition.

lower triangular matrices  $L_i$ ,

$$\underbrace{L_{n-1} \cdots L_2 L_1}_L A = U. \quad (\text{B.2})$$

After completing these operations, we have  $U$  and we can find  $L$  by computing  $L = L_1^{-1} L_2^{-1} \cdots L_{n-1}^{-1}$ .

Once we have this decomposition, we have  $LUu = b$ . Setting  $Uu$  to  $y$ , we can solve  $Ly = b$  for  $y$  by forward substitution. Now we have  $Uu = y$ , which we can solve by back substitution for  $u$ .

---

**Algorithm B.1:** Solving  $Au = b$  by LU decomposition

---

**Inputs:**

$A$ : Nonsingular square matrix

$b$ : A vector

**Outputs:**

$u$ : Solution to  $Au = b$

---

Perform forward substitution to solve  $Ly = b$  for  $y$ :

$$y_1 = \frac{b_1}{L_{11}}, \quad y_i = \frac{1}{L_{ii}} \left( b_i - \sum_{j=1}^{i-1} L_{ij} y_j \right) \quad \text{for } i = 2, \dots, n$$

Perform backward substitution to solve the following  $Uu = y$  for  $u$ :

$$u_n = \frac{y_n}{U_{nn}}, \quad u_i = \frac{1}{U_{ii}} \left( y_i - \sum_{j=i+1}^n U_{ij} u_j \right) \quad \text{for } i = n-1, \dots, 1$$


---

The process described above is not stable in general and needs to be modified. In particular, roundoff errors are magnified in the backward substitution when diagonal elements of  $A$  have a small magnitude. This issue is resolved by using *partial pivoting*, which interchanges rows to obtain more favorable diagonal elements.

*Cholesky decomposition* is an LU decomposition specialized for the case where the matrix  $A$  is symmetric and positive definite. In this case, pivoting is not necessary because the Gaussian elimination is always stable for symmetric positive definite matrices. The decomposition can be written as

$$A = LDL^T, \quad (\text{B.3})$$

where  $D = \text{diag}[U_{11}, \dots, U_{nn}]$ . This can be expressed as the matrix product

$$A = GG^T, \quad (\text{B.4})$$

where  $G = LD^{1/2}$ .

## B.2 Iterative Methods

While direct methods are usually more efficient and robust, iterative methods have several advantages:

- Iterative methods make it possible to trade between computational cost and precision because they can be stopped at any point and still yield an approximation of  $u$ . Direct methods, on the other hand, only get the solution at the end of the process with the final precision.
- Iterative methods have the advantage when a good guess for  $u$  exists. This is often the case in optimization, where the  $u$  from the previous optimization iteration can be used as the guess for the new evaluations (called a *warm start*).
- Iterative methods do not require forming and manipulating the matrix  $A$ , which can be computational costly in terms of both time and memory. Instead, iterative methods require the computation of the residuals  $r(u) = Au - b$  and in the case of Krylov subspace methods, products of  $A$  with a given vector. Therefore, iterative methods can be more efficient than direct methods for cases where  $A$  is large and sparse. All that is needed is an efficient process to get the product of  $A$  with a given vector, as shown in Fig. B.2

Iterative methods are divided into fixed-point iteration methods (also known as stationary iterative methods) and Krylov subspace methods.



**Figure B.2:** Iterative methods just require a process to compute products of  $A$  with an arbitrary vector  $v$ .

### Fixed-point Iteration Methods

Fixed-point methods generate a sequence of iterates  $u^{(1)}, \dots, u^{(k)}, \dots$  using a function

$$u^{(k+1)} = G(u^{(k)}), \quad k = 0, 1, \dots, \quad (\text{B.5})$$

starting from an initial guess  $u_0$ . The function  $G(u)$  is devised such that the iterates converge to the solution  $u^*$ , which satisfies  $r(u^*) = 0$ . Many fixed-point methods can be derived by *splitting* the matrix such that  $A = M - N$ . Then,  $Au = b$  leads to  $Mu = Nu + b$ , and substituting this into the linear system yields

$$u = M^{-1}(Nu + b). \quad (\text{B.6})$$

Since  $Nu = Mu - Au$ , substituting this in to the above equation results in the iteration

$$u^{(k+1)} = u^{(k)} + M^{-1} \left( b - Au^{(k)} \right) = u^{(k)} + M^{-1} r \left( u^{(k)} \right). \quad (\text{B.7})$$

The splitting matrix  $M$  is fixed and constructed so that it is easy to invert. The closer  $M^{-1}$  is to the inverse of  $A$ , the better the iterations work. We now introduce three fixed-point methods corresponding to three different splitting matrices.

The Jacobi method consists of setting  $M$  to be a diagonal matrix  $D$  where the diagonal entries are those of  $A$ . Then,

$$u^{(k+1)} = u^{(k)} + D^{-1} r \left( u^{(k)} \right). \quad (\text{B.8})$$

In component form, this can be written as

$$u_i^{(k+1)} = \frac{1}{A_{ii}} \left[ b_i - \sum_{j=1, j \neq i} A_{ij} u_j^{(k)} \right], \quad i = 1, \dots, n_u \quad (\text{B.9})$$

Using this method, each component in  $u_{k+1}$  is independent of each other at a given iteration; they only depend on the previous iteration values,  $u_k$ , and can therefore be done in parallel.

The Gauss–Seidel method is obtained by setting  $M$  to be the lower triangular portion of  $A$ , and can be written as

$$u_{k+1} = u_k + E^{-1} R(u_k), \quad (\text{B.10})$$

where  $E$  is the lower triangular matrix. Because of the triangular matrix structure, each component in  $u_{k+1}$  is dependent on the previous components in the vector, but the iteration can be performed in a single forward sweep. Writing this in component form yields

$$u_i^{(k+1)} = \frac{1}{A_{ii}} \left[ b_i - \sum_{j < i} A_{ij} u_j^{(k+1)} - \sum_{j > i} A_{ij} u_j^{(k)} \right], \quad i = 1, \dots, n_u. \quad (\text{B.11})$$

Unlike the Jacobi iterations, a Gauss–Seidel iteration cannot be performed in parallel because of the terms where  $j < i$  above, which require the latest values. Instead, the states must be updated sequentially. However, the advantage of Gauss–Seidel is that it generally converges faster than Jacobi iterations.

The successive over-relaxation (SOR) method uses an update that is a weighted average of the Gauss–Seidel update and the previous iteration,

$$u_{k+1} = u_k + \omega [(1 - \omega) D + \omega E]^{-1} R(u_k), \quad (\text{B.12})$$

where  $\omega$  is a scalar between one and two. Setting  $\omega = 1$  above yields the Gauss–Seidel method. SOR in component form is

$$u_i^{(k+1)} = (1-\omega)u_i^{(k)} + \frac{\omega}{A_{ii}} \left[ b_i - \sum_{j < i} A_{ij}u_j^{(k+1)} - \sum_{j > i} A_{ij}u_j^{(k)} \right], \quad i = 1, \dots, n_u. \quad (\text{B.13})$$

With the right value of  $\omega$ , SOR converges faster than Gauss–Seidel.

---

**Example B.2:** Iterative methods applied to a simple linear system.

Suppose we have a linear system of two equations

$$\begin{bmatrix} 2 & -1 \\ -2 & 3 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \quad (\text{B.14})$$

This corresponds to the two lines shown in Fig. B.3, where the solution is at their intersection.

Applying the Jacobian iteration (B.9),

$$\begin{aligned} u_1^{(k+1)} &= \frac{1}{2}u_2^{(k)} \\ u_2^{(k+1)} &= \frac{1}{3}(1 + 2u_1^{(k)}). \end{aligned} \quad (\text{B.15})$$

Starting with the guess  $u^{(0)} = (2, 1)$ , we get the iterations shown in Fig. B.3. The Gauss–Seidel iteration (B.11) is similar, where the only change is that the second equation use the latest state from the first one:

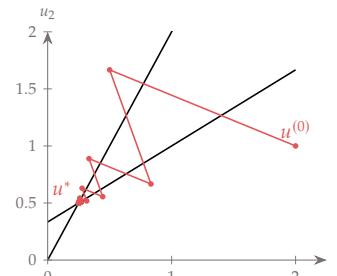
$$\begin{aligned} u_1^{(k+1)} &= \frac{1}{2}u_2^{(k)} \\ u_2^{(k+1)} &= \frac{1}{3}(1 + 2u_1^{(k+1)}). \end{aligned} \quad (\text{B.16})$$

As expected, Gauss–Seidel converges faster than the Jacobi iteration, taking a more direct path. The SOR iteration is

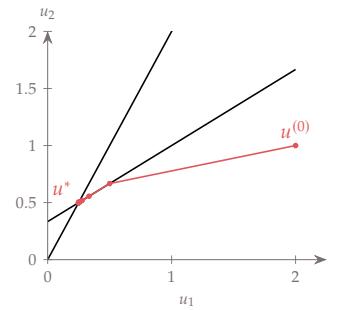
$$\begin{aligned} u_1^{(k+1)} &= (1 - \omega)u_1^{(k)} + \frac{\omega}{2}u_2^{(k)} \\ u_2^{(k+1)} &= (1 - \omega)u_2^{(k)} + \frac{\omega}{3}(1 + 2u_1^{(k)}). \end{aligned} \quad (\text{B.17})$$

SOR converges even faster for the right values of  $\omega$ . The result shown here is for  $\omega = 1.2$ .

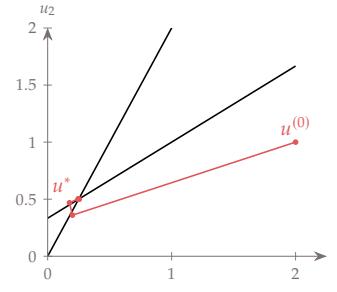
---



(a) Jacobi



(b) Gauss-Seidel



(c) SOR

**Figure B.3:** Jacobi, Gauss–Seidel, and SOR iterations.

## Krylov Subspace Methods

Krylov subspace methods are another class of iterative methods that include the conjugate gradient method and the generalized minimum residual (GMRES) method. Compared to stationary methods,

Krylov methods have the advantage that they use information gathered throughout the iterations. Instead of using a fixed splitting matrix, Krylov methods effectively vary the splitting so that  $M$  is changed at each iteration according to some criteria that uses the information gathered so far. For this reason, Krylov methods are usually more efficient than fixed-point iterations.

Like fixed-point iteration methods, Krylov methods do not require forming or storing  $A$ . Instead, the iterations require only matrix-vector products of the form  $Av$ , where  $v$  is some vector given by the Krylov algorithm. To be efficient, Krylov subspace methods require a good *preconditioner*.

## Test Problems

# C

### C.1 Unconstrained Problems

#### C.1.1 Slanted Quadratic Function

This is a smooth two-dimensional function suitable for a first test of a gradient-based optimizer:

$$f(x_1, x_2) = x_1^2 + x_2^2 - \beta x_1 x_2, \quad (\text{C.1})$$

where  $\beta \in [0, 2)$ . A  $\beta$  value of zero corresponds to perfectly circular contours. As  $\beta$  increases, the contours become increasingly slanted. For  $\beta = 2$ , the quadratic becomes semidefinite and there is a line of weak minima. For  $\beta > 2$ , the quadratic is indefinite and there is no minimum. An intermediate value of  $\beta = 3/2$  is suitable for first tests and yields the contours shown in Fig. C.1.

*Global minimum:*  $f(x^*) = 0.0$  at  $x^* = (0, 0)$

#### C.1.2 Rosenbrock Function

The two-dimensional Rosenbrock function, shown in Fig. C.2, is:

$$f(x_1, x_2) = (1 - x_1)^2 + 100 (x_2 - x_1^2)^2. \quad (\text{C.2})$$

This is a classic benchmarking function because of its narrow turning valley. The large difference between the maximum and minimum curvatures, and the fact that the principal curvature directions change along the valley makes it a good test for quasi-Newton methods.

The Rosenbrock function can be extended to  $n$ -dimensions by defining the sum,

$$f(x) = \sum_{i=1}^{n-1} \left( 100 (x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \right). \quad (\text{C.3})$$

*Global minimum:*  $f(x^*) = 0.0$  at  $x^* = (1, 1, \dots, 1)$ .

*Local minimum:* For  $n \geq 4$ , a local minimum exists near  $x = (-1, 1, \dots, 1)$ .

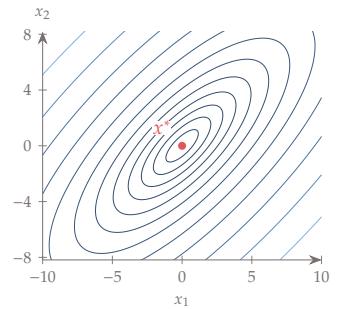


Figure C.1: Slanted quadratic function for  $\beta = 3/2$

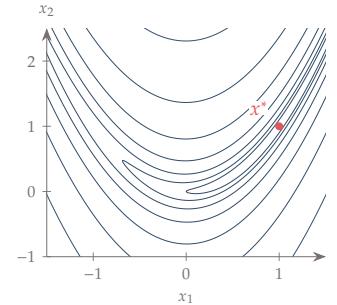


Figure C.2: Rosenbrock function

### C.1.3 Bean Function

The “bean” function was developed in this book as a milder version of the Rosenbrock function: it has the same curved valley as the Rosenbrock function without the extreme variations in curvature. The function, shown in Fig. C.3, is

$$f(x_1, x_2) = (1 - x_1)^2 + (1 - x_2)^2 + \frac{1}{2} (2x_2 - x_1^2)^2. \quad (\text{C.4})$$

*Global minimum:*  $f(x^*) = 0.09194$  at  $x^* = (1.21314, 0.82414)$

### C.1.4 Jones Function

This is a fourth-order smooth multimodal function that is useful to test global search algorithms and also gradient-based algorithms starting from different points. There are saddle points, maxima, and minima, with one global minimum. This function, shown in Fig. C.4 along with the local and global minima, is

$$f(x_1, x_2) = x_1^4 + x_2^4 - 4x_1^3 - 3x_2^3 + 2x_1^2 + 2x_1x_2. \quad (\text{C.5})$$

*Global minimum:*  $f(x^*) = -13.5320$  at  $x^* = (2.6732, -0.6759)$

*Local minimum:*  $f(x) = -9.7770$  at  $x = (-0.4495, 2.2928)$

$f(x) = -9.0312$  at  $x = (2.4239, 1.9219)$

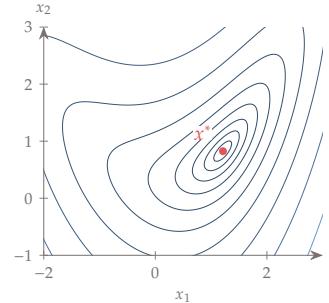


Figure C.3: Bean function

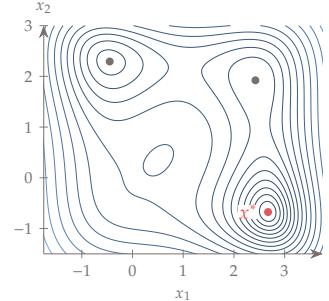


Figure C.4: Jones multimodal function

### C.1.5 Hartmann Function

The Hartmann function is a three-dimensional smooth function with multiple local minima.

$$f(x) = -\sum_{i=1}^4 \alpha_i \exp\left(-\sum_{j=1}^3 A_{ij}(x_j - P_{ij})^2\right) \quad (\text{C.6})$$

where

$$\begin{aligned} \alpha &= [1.0, 1.2, 3.0, 3.2]^T, \\ A &= \begin{bmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{bmatrix}, \\ P &= 10^{-4} \begin{bmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{bmatrix}. \end{aligned} \quad (\text{C.7})$$

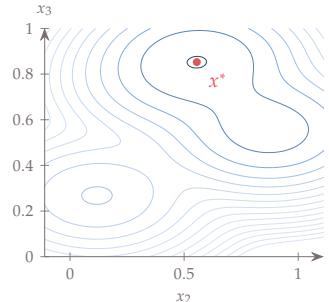


Figure C.5: An  $x_2 - x_3$  slice of Hartmann function at  $x_1 = 0.1148$

A slice of the function, at the optimal value of  $x_1 = 0.1148$ , is shown in Fig. C.5.

*Global minimum:  $f(x^*) = -3.86278$  at  $x^* = (0.11480, 0.55566, 0.85254)$*

### C.1.6 Aircraft Wing Design

We want to optimize the wing of a general aviation sized aircraft by changing its [wing span](#) and [chord](#). In general, we would add many more design variables to a problem like this, but we are intentionally limiting it to a simple two-dimensional problem so we can easily visualize the results. Instead of minimizing drag, we want to minimize the required power, thereby taking into account drag as well as propulsive efficiency that is speed dependent.

The following section describes a basic performance estimation methodology for a low-speed aircraft. Implementing it may not seem like it has much to do with optimization. The physics are important for our purposes, but practice translating equations and concepts into code is an important element of formulating optimization problems in general.

At level flight the aircraft must generate enough lift to equal the required weight

$$L = W, \quad (\text{C.8})$$

and we will assume here that the total weight consists of a fixed aircraft and payload weight  $W_0$ , and a component of the weight that depends on the wing area  $S$

$$W = W_0 + W_s S. \quad (\text{C.9})$$

Our wing can produce a certain lift coefficient,  $C_L$ , and so we must make the wing area,  $S$ , big enough to produce sufficient lift. Using the definition of lift coefficient, the total lift can be computed as

$$L = q C_L S, \quad (\text{C.10})$$

where  $q$  is the dynamic pressure

$$q = \frac{1}{2} \rho v^2. \quad (\text{C.11})$$

If we use a rectangular wing, then the wing area can be computed from the wing span,  $b$ , and the chord,  $c$ , as

$$S = bc. \quad (\text{C.12})$$

The drag of our aircraft consists of two components: viscous drag and induced drag. The viscous drag can be approximated as

$$D_f = k C_f q S_{\text{wet}}. \quad (\text{C.13})$$

For a fully turbulent boundary layer, the skin friction coefficient,  $C_f$ , can be approximated as

$$C_f = \frac{0.074}{Re^{0.2}}. \quad (\text{C.14})$$

In this equation the Reynolds number is based on the wing chord,  $Re = \rho v c / \mu$ . The form factor,  $k$ , accounts for the effects of pressure drag. The wetted area,  $S_{\text{wet}}$ , is the area over which the skin friction drag acts, which is a little more than twice the planform area. We will use

$$S_{\text{wet}} = 2.05S. \quad (\text{C.15})$$

The induced drag is defined as

$$D_i = \frac{L^2}{q\pi b^2 e}, \quad (\text{C.16})$$

where  $e$  is the Oswald efficiency factor. Total drag is the sum of induced and viscous drag,  $D = D_i + D_f$ .

Our objective function, the power required by the motor for level flight, is

$$P(b, c) = \frac{Dv}{\eta}, \quad (\text{C.17})$$

where  $\eta$  is the propulsive efficiency. We assume that our electric propellers have a Gaussian efficiency curve (real efficiency curves aren't very Gaussian, but this is simple and will be sufficient for our purposes):

$$\eta = \eta_{\max} \exp\left(\frac{-(v - \bar{v})^2}{2\sigma^2}\right). \quad (\text{C.18})$$

In this problem, the lift coefficient is provided. Therefore, to satisfy the lift requirement (C.8), we need to compute the velocity using Eq. C.11 and Eq. C.10 as

$$v = \sqrt{\frac{2L}{\rho C_L S}}. \quad (\text{C.19})$$

The parameters for this problem are given as:

Parameter	Value	Unit	Description
$\rho$	1.2	$\text{kg}/\text{m}^3$	density of air
$\mu$	$1.8 \times 10^{-5}$	$\text{kg}/(\text{m sec})$	viscosity of air
$k$	1.2		form factor
$C_L$	0.4		lift coefficient
$e$	0.80		Oswald efficiency factor
$W$	1,000	N	fixed aircraft weight
$W_S$	8.0	$\text{N}/\text{m}^2$	wing area dependent weight
$\eta_{\max}$	0.8		peak propulsive efficiency
$\bar{v}$	20.0	m/s	flight speed at peak
$\sigma$	5.0	m/s	propulsive efficiency standard deviation of efficiency function

This is the same problem that was presented in Ex. 1.2 of Chapter 1. The optimal wing span and chord are  $b = 25.48$  m and  $c = 0.50$  m respectively given the parameters. The contour and the optimal wing shape are shown in Fig. C.6.

Note that there are no structural considerations in this problem so the resulting aircraft has a higher aspect ratio wing than is realistic. This emphasizes the importance of carefully selecting the objective, and including all relevant constraints.

### C.1.7 Brachistochrone Problem

This is the classic problem proposed by Johann Bernoulli (see Section 2.2 for the historical background). Although this was originally solved analytically, we discretize the model and solve the problem using numerical optimization. This is a useful problem for benchmarking because you can change the number of dimensions.

A bead is set on wire that defines a path that we can shape. The bead starts at some  $y$ -position  $h$  with zero velocity. For convenience, we define the starting point at  $x = 0$ . From conservation of energy, we can then find the velocity of the bead at any other location. The initial potential energy is converted to kinetic energy, potential energy, and dissipative work from friction acting along the path length, yielding:

$$\begin{aligned} mgh &= \frac{1}{2}mv^2 + mgy + \int_0^x \mu_k mg \cos \theta \, ds, \\ 0 &= \frac{1}{2}v^2 + g(y - h) + \mu_k gx, \\ v &= \sqrt{2g(h - y - \mu_k x)}. \end{aligned} \tag{C.20}$$

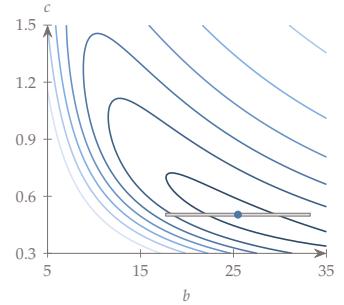


Figure C.6: Wing design problem with power requirement contour

Now that we know the speed of the bead as a function of  $x$ , we can compute the time it takes to traverse an infinitesimal element of length  $ds$ :

$$\begin{aligned}\Delta t &= \int_{x_i}^{x_i+dx} \frac{ds}{v(x)} \\ &= \int_{x_i}^{x_i+dx} \frac{\sqrt{dx^2 + dy^2}}{\sqrt{2g(h - y(x) - \mu_k x)}} \\ &= \int_{x_i}^{x_i+dx} \frac{\sqrt{1 + \left(\frac{dy}{dx}\right)^2} dx}{\sqrt{2g(h - y(x) - \mu_k x)}}\end{aligned}\quad (\text{C.21})$$

To discretize this problem, we can divide the path into linear segments. As an example, Fig. C.7 shows the wire divided into 4 linear segments (5 nodes) as an approximation of a continuous wire. The slope  $s_i = (\Delta y / \Delta x)_i$  is then a constant along a given segment, and  $y(x) = y_i + s_i(x - x_i)$ . Making these substitutions results in

$$\Delta t_i = \frac{\sqrt{1 + s_i^2}}{\sqrt{2g}} \int_{x_i}^{x_{i+1}} \frac{dx}{\sqrt{h - y_i - s_i(x - x_i) - \mu_k x}}. \quad (\text{C.22})$$

Performing the integration and simplifying (many steps omitted here) results in

$$\Delta t_i = \sqrt{\frac{2}{g} \frac{\sqrt{\Delta x_i^2 + \Delta y_i^2}}{\sqrt{h - y_{i+1} - \mu_k x_{i+1}} + \sqrt{h - y_i - \mu_k x_i}}}, \quad (\text{C.23})$$

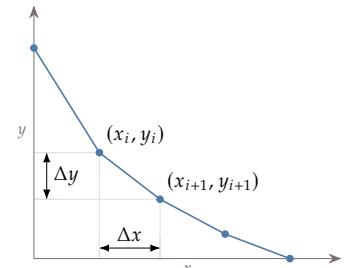
where  $\Delta x_i = (x_{i+1} - x_i)$  and  $\Delta y_i = (y_{i+1} - y_i)$ . The objective of the optimization is to minimize the total travel time, so we need to sum up the travel time across all of our linear segments,

$$T = \sum_{i=1}^{n-1} \Delta t_i. \quad (\text{C.24})$$

Minimization is unaffected by multiplying by a constant, so we can remove the multiplicative constant for simplicity (we see that the magnitude of the acceleration of gravity has no affect on the optimal path)

$$\text{minimize } f = \sum_{i=1}^{n-1} \frac{\sqrt{\Delta x_i^2 + \Delta y_i^2}}{\sqrt{h - y_{i+1} - \mu_k x_{i+1}} + \sqrt{h - y_i - \mu_k x_i}} \quad (\text{C.25})$$

by varying  $y_i$ ,  $i = 1, \dots, n$



**Figure C.7:** A discretized representation of the brachistochrone problem.

The design variables are the  $n - 2$  positions of the path parameterized by  $y_i$ . The end points must be fixed, otherwise the problem is ill-defined, which is why there are  $n - 2$  design variables instead of  $n$ . Note that  $x$  is a parameter, meaning that it is fixed. You could space the  $x_i$  any reasonable way and still find the same underlying optimal curve, but it is easiest to just use uniform spacing. As the dimensionality of the problem increases, the solution becomes more challenging. We will use the following specifications:

- starting point:  $(x, y) = (0, 1)$  m
- ending point:  $(x, y) = (1, 0)$  m
- kinetic coefficient of friction  $\mu_k = 0.3$

The analytic solution for the case with friction is more difficult to derive, but the analytic solution for the frictionless case ( $\mu_k = 0$ ) with our starting and ending points is:

$$\begin{aligned} x &= a(\theta - \sin(\theta)), \\ y &= -a(1 - \cos(\theta)) + 1, \end{aligned} \tag{C.26}$$

where  $a = 0.572917$  and  $\theta \in [0, 2.412]$ .

## C.2 Constrained Problems

### C.2.1 Barnes Problem

The Barnes problem was devised in an M.S. Thesis<sup>148</sup> and has been used in various optimization demonstration studies. It is a nice starter problem because it only has two dimensions for easy visualization while also including constraints. The objective function contains the following coefficients:

$$\begin{array}{ll} a_1 = 75.196 & a_2 = -3.8112 \\ a_3 = 0.12694 & a_4 = -2.0567 \times 10^{-3} \\ a_5 = 1.0345 \times 10^{-5} & a_6 = -6.8306 \\ a_7 = 0.030234 & a_8 = -1.28134 \times 10^{-3} \\ a_9 = 3.5256 \times 10^{-5} & a_{10} = -2.266 \times 10^{-7} \\ a_{11} = 0.25645 & a_{12} = -3.4604 \times 10^{-3} \\ a_{13} = 1.3514 \times 10^{-5} & a_{14} = -28.106 \\ a_{15} = -5.2375 \times 10^{-6} & a_{16} = -6.3 \times 10^{-8} \\ a_{17} = 7.0 \times 10^{-10} & a_{18} = 3.4054 \times 10^{-4} \\ a_{19} = -1.6638 \times 10^{-6} & a_{20} = -2.8673 \\ a_{21} = 0.0005 & \end{array}$$

<sup>148</sup> Barnes, *A Comparative Study of Non-linear Optimization Codes*. 1967

And for convenience we define the following quantities:

$$y_1 = x_1 x_2, \quad y_2 = y_1 x_1, \quad y_3 = x_2^2, \quad y_4 = x_1^2 \quad (\text{C.27})$$

The objective function is then:

$$\begin{aligned} f(x_1, x_2) = & a_1 + a_2 x_1 + a_3 y_4 + a_4 y_4 x_1 + a_5 y_4^2 + a_6 x_2 + a_7 y_1 + \\ & a_8 x_1 y_1 + a_9 y_1 y_4 + a_{10} y_2 y_4 + a_{11} y_3 + a_{12} x_2 y_3 + a_{13} y_3^2 + \\ & \frac{a_{14}}{x_2 + 1} + a_{15} y_3 y_4 + a_{16} y_1 y_4 x_2 + a_{17} y_1 y_3 y_4 + a_{18} x_1 y_3 + \\ & a_{19} y_1 y_3 + a_{20} \exp(a_{21} y_1) \end{aligned} \quad (\text{C.28})$$

There are three constraints of the form  $g(x) \leq 0$ :

$$\begin{aligned} g_1 &= 1 - \frac{y_1}{700} \\ g_2 &= \frac{y_4}{25^2} - \frac{x_2}{5} \\ g_3 &= \frac{x_1}{500} - 0.11 - \left( \frac{x_2}{50} - 1 \right)^2 \end{aligned} \quad (\text{C.29})$$

The problem also has bound constraints. The original formulation is bounded from  $[0, 80]$  in both dimensions, in which case the global optimum occurs in the corner at  $x^* = [80, 80]$  with a local minimum in the middle. However, in our usage we preferred the global optimum to not be in the corner so set the bounds to  $[0, 65]$  for  $x_1$  and  $[0, 70]$  for  $x_2$ . The contour of this function is plotted in Fig. C.8.

*Global minimum:*  $f(x^*) = -31.6368$  at  $x^* = (49.5263, 19.6228)$

*Local minimum:*  $f(x) = -17.7067$  at  $x = (65, 70)$

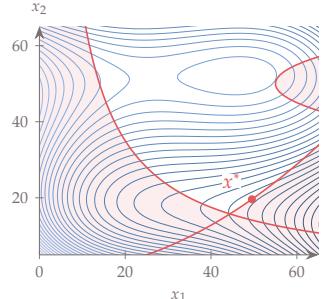


Figure C.8: Barnes function

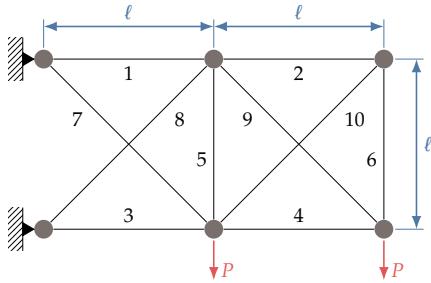
C.2.2 Ten-bar Truss

The ten bar truss is a classic optimization problem.<sup>149</sup> In this problem, we want to find the optimal cross-sectional areas for the ten-bar truss shown in Fig. C.9. A simple truss finite element code set up for this particular configuration is available in the book code repository. The function takes in an array of cross-sectional areas and returns the total mass and an array of stresses for each truss member.

The objective of the optimization is to minimize the mass of the structure, subject to the constraints that every segment does not yield in compression or tension. The yield stress of all elements is  $25 \times 10^3$  psi, except for member 9, which uses a stronger alloy with a yield stress of  $75 \times 10^3$  psi. Mathematically, the constraint is

$$|\sigma_i| \leq \sigma_y, \quad \text{for } i = 1, \dots, 10, \quad (\text{C.30})$$

<sup>149</sup>. Venkayya, *Design of optimum structures*. 1971



**Figure C.9:** Ten-bar truss and element numbers.

where the absolute value is needed to handle tension and compression (with the same yield strength for tension and compression). Absolute values are not differentiable at 0 and should be avoided in gradient-based optimization if possible. Put this in a mathematically equivalent form that avoids absolute value. Each element should have a cross-sectional area of at least 0.1 in<sup>2</sup> for manufacturing reasons (bound constraint). In solving this optimization problem you may need to consider scaling the objective and constraints.

While not needed to solve the problem, an overview of the equations is discussed below. A truss element is the simplest type of finite element and only has an axial degree of freedom. The theory and derivation for truss elements is simple, but for our purposes we will jump to the result. Given a 2D element oriented arbitrarily in space (Fig. C.10) we can relate the displacements at the nodes to the forces at the nodes through a stiffness relationship.

In matrix form the equation for a given element is  $f = K_e x$ . In detail the equation is

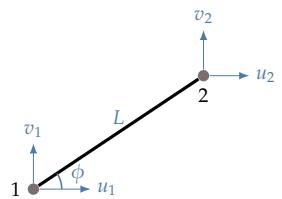
$$\begin{bmatrix} X_1 \\ Y_1 \\ X_2 \\ Y_2 \end{bmatrix} = \frac{EA}{L} \begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix} \quad (\text{C.31})$$

where the displacement vector is  $x = [u_1, v_1, u_2, v_2]^T$ . The meanings for the variables in the equation are described in Table C.1.

The stress in the truss element can be computed from the equation  $\sigma = S_e x$  where  $\sigma$  is a scalar,  $x$  is the same vector as before, and the element  $S_e$  matrix (really a row vector because stress is one-dimensional for truss elements) is:

$$S_e = \frac{E}{L} [-c \quad -s \quad c \quad s] \quad (\text{C.32})$$

The global structure (an assembly of multiple finite elements) has the same equations:  $F = Kx$  and  $\sigma = Sx$ , but now  $x$  contains displacements



**Figure C.10:** A truss element oriented at some angle  $\phi$ , where  $\phi$  is measured from a horizontal line emanating from the first node, oriented in the + $x$  direction.

**Table C.1:** The variables used in the stiffness equation.

Variable	Description
$X_i$	force in the x-direction at node i
$Y_i$	force in the y-direction at node i
$E$	modulus of elasticity of truss element material
$A$	area of truss element cross-section
$L$	length of truss element
$c$	$\cos \phi$
$s$	$\sin \phi$
$u_i$	displacement in the x-direction at node i
$v_i$	displacement in the y-direction at node i

for all of the nodes in the structure  $x = [x_1, x_2, \dots, x_n]^T$ . If we have  $n$  nodes and  $m$  elements then  $F \in \mathcal{R}^{2n}$ ,  $x \in \mathcal{R}^{2n}$ ,  $K \in \mathcal{R}^{2n \times 2n}$ ,  $S \in \mathcal{R}^{m \times 2n}$ , and  $\sigma \in \mathcal{R}^m$ . The elemental stiffness and stress matrices are first computed and then assembled into the global matrices. This is straightforward because the displacements and forces of the individual elements add linearly.

After we assemble the global matrices we must remove any degrees of freedom that are structurally rigid (already known to have zero displacement). Otherwise, the problem is ill-defined and the stiffness matrix will be ill-conditioned.

Given the geometry, materials, and external loading we can populate the stiffness matrix and force vector. We can then solve for the unknown displacements from

$$F = Kx \quad (\text{C.33})$$

With the solved displacements we can compute the stress in each element using

$$\sigma = Sx \quad (\text{C.34})$$

## Bibliography

---

- 1 Bryson, A. E. and Ho, Y. C., *Applied Optimal Control; Optimization, Estimation, and Control*. Blaisdell Publishing, 1969. cited on p. 24
- 2 Bertsekas, D. P., *Dynamic programming and optimal control*. Belmont, MA: Athena Scientific, 1995. cited on p. 24
- 3 Kepler, J., *Nova stereometria doliorum vinariorum (New solid geometry of wine barrels)*. Linz: Johannes Planck, 1615. cited on p. 32
- 4 Ferguson, T. S., “Who Solved the Secretary Problem?” *Statistical Science*, Vol. 4, No. 3, August 1989, pp. 282–289. cited on p. 32  
doi: 10.1214/ss/1177012493
- 5 Fermat, P. de, *Methodus ad disquirendam maximam et minimam (Method for the study of maxima and minima)*. 1636, Translated by Jason Ross. cited on p. 33
- 6 Lagrange, J.-L., *Mécanique analytique*. Paris, France, 1788, Vol. 1. cited on p. 34
- 7 Cauchy, A.-L., “Méthode générale pour la résolution des systèmes d’équations simultanées,” *Comptes rendus hebdomadaires des séances de l’Académie des sciences*, Vol. 25, October 1847, pp. 536–538. cited on p. 34
- 8 Hancock, H., *Theory of Minima and Maxima*. Boston, MA: Ginn and Company, 1917. cited on p. 34
- 9 Menger, K., “Das botenproblem,” *Ergebnisse eines Mathematischen Kolloquiums*, Leipzig: Teubner, 1932, pp. 11–12. cited on p. 34
- 10 Karush, W., “Minima of Functions of Several Variables with Inequalities as Side Constraints,” Master’s thesis, University of Chicago, Chicago, IL, 1939. cited on p. 35
- 11 Krige, D. G., “A statistical approach to some mine valuation and allied problems on the Witwatersrand,” Master’s thesis, University of the Witwatersrand, Johannesburg, South Africa, 1951. cited on p. 35
- 12 Markowitz, H., “Portfolio selection,” *Journal of Finance*, Vol. 7, March 1952, pp. 77–91. cited on p. 35

- 13 Davidon, W. C., "Variable Metric Method for Minimization," *SIAM Journal on Optimization*, Vol. 1, No. 1, February 1991, pp. 1–17, ISSN: 1095-7189.  
doi: [10.1137/0801001](https://doi.org/10.1137/0801001) cited on pp. 36, 102
- 14 Fletcher, R. and Powell, M. J. D., "A Rapidly Convergent Descent Method for Minimization," *The Computer Journal*, Vol. 6, No. 2, August 1963, pp. 163–168, ISSN: 1460-2067.  
doi: [10.1093/comjnl/6.2.163](https://doi.org/10.1093/comjnl/6.2.163) cited on pp. 36, 102
- 15 Wolfe, P., "Convergence Conditions for Ascent Methods," *SIAM Review*, Vol. 11, No. 2, 1969, pp. 226–235.  
doi: [10.1137/1011036](https://doi.org/10.1137/1011036) cited on p. 36
- 16 Wilson, R. B., "A simplicial algorithm for concave programming," Ph.D. Dissertation, Harvard University, Cambridge, MA, June 1963. cited on p. 36
- 17 Han, S.-P., "Superlinearly convergent variable metric algorithms for general nonlinear programming problems," *Mathematical Programming*, Vol. 11, No. 1, 1976, pp. 263–282.  
doi: [10.1007/BF01580395](https://doi.org/10.1007/BF01580395) cited on p. 36
- 18 Powell, M. J. D., "Algorithms for nonlinear constraints that use Lagrangian functions," *Mathematical Programming*, Vol. 14, No. 1, December 1978, pp. 224–248.  
doi: [10.1007/bf01588967](https://doi.org/10.1007/bf01588967) cited on pp. 36, 153
- 19 Holland, J. H., "Aptation in Natural and Artificial Systems," Ann Arbor, MI: University of Michigan Press, 1975. cited on p. 36
- 20 Hooke, R. and Jeeves, T. A., "“Direct Search” Solution of Numerical and Statistical Problems," *Journal of the ACM*, Vol. 8, No. 2, 1961, pp. 212–229. cited on p. 37
- 21 Nelder, J. A. and Mead, R., "A Simplex Method for Function Minimization," *Computer Journal*, Vol. 7, 1965, pp. 308–313. cited on pp. 37, 217
- 22 Karmarkar, N., "A New Polynomial-Time Algorithm for Linear Programming," *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '84, New York, NY, USA: Association for Computing Machinery, 1984, pp. 302–311, ISBN: 0897911334  
doi: [10.1145/800057.808695](https://doi.org/10.1145/800057.808695) cited on p. 37
- 23 Pontryagin, L. S., Boltyanskii, V. G., Gamkrelidze, R. V., and Mishchenko, E. F., *The Mathematical Theory of Optimal Processes*. Moscow, 1961, translated by K. N. Trirogoff, edited by T. W. Neustadt. cited on p. 37

- 24 Bryson Jr, A. E., "Optimal Control—1950 to 1985," *IEEE Control Systems Magazine*, Vol. 16, No. 3, June 1996, pp. 26–33. cited on p. 37  
DOI: [10.1109/37.506395](https://doi.org/10.1109/37.506395)
- 25 Schmit, L. A., "Structural Design by Systematic Synthesis," *Proceedings of the 2nd National Conference on Electronic Computation*, ASCE, New York, NY, September 1960, pp. 105–132. cited on pp. 37, 170
- 26 Schmit, L. A. and Thornton, W. A., "Synthesis of an Airfoil at Supersonic Mach Number," CR 144, NASA, January 1965. cited on p. 38
- 27 Fox, R. L., "Constraint Surface Normals for Structural Synthesis Techniques," *AIAA Journal*, Vol. 3, No. 8, August 1965, pp. 1517–1518. cited on p. 38  
DOI: [10.2514/3.3182](https://doi.org/10.2514/3.3182)
- 28 Arora, J. and Haug, E. J., "Methods of Design Sensitivity Analysis in Structural Optimization," *AIAA Journal*, Vol. 17, No. 9, 1979, pp. 970–974. cited on p. 38  
DOI: [10.2514/3.61260](https://doi.org/10.2514/3.61260)
- 29 Haftka, R. T. and Grandhi, R. V., "Structural shape optimization—A survey," *Computer Methods in Applied Mechanics and Engineering*, Vol. 57, No. 1, 1986, pp. 91–106, ISSN: 0045-7825. cited on p. 38  
DOI: [10.1016/0045-7825\(86\)90072-1](https://doi.org/10.1016/0045-7825(86)90072-1)
- 30 Eschenauer, H. A. and Olhoff, N., "Topology optimization of continuum structures: A review," *Applied Mechanics Reviews*, Vol. 54, No. 4, July 2001, pp. 331–390. cited on p. 38  
DOI: [10.1115/1.1388075](https://doi.org/10.1115/1.1388075)
- 31 Pironneau, O., "On optimum design in fluid mechanics," *Journal of Fluid Mechanics*, Vol. 64, No. 01, 1974, p. 97, ISSN: 0022-1120. cited on p. 38  
DOI: [10.1017/S0022112074002023](https://doi.org/10.1017/S0022112074002023)
- 32 Jameson, A., "Aerodynamic Design via Control Theory," *Journal of Scientific Computing*, Vol. 3, No. 3, September 1988, pp. 233–260. cited on p. 38  
DOI: [10.1007/BF01061285](https://doi.org/10.1007/BF01061285)
- 33 Sobieszczanski-Sobieski, J. and Haftka, R. T., "Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments," *Structural Optimization*, Vol. 14, No. 1, 1997, pp. 1–23. cited on p. 38  
DOI: [10.1007/BF011](https://doi.org/10.1007/BF011)
- 34 Martins, J. R. R. A. and Lambe, A. B., "Multidisciplinary Design Optimization: A Survey of Architectures," *AIAA Journal*, Vol. 51, No. 9, September 2013, pp. 2049–2075. cited on pp. 38, 375, 395  
DOI: [10.2514/1.J051895](https://doi.org/10.2514/1.J051895)

- 35 Sobiesczanski-Sobieski, J., "Sensitivity of Complex, Internally Coupled Systems," *AIAA Journal*, Vol. 28, No. 1, 1990, pp. 153–160.  
doi: [10.2514/3.10366](https://doi.org/10.2514/3.10366) cited on p. 38
- 36 Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J., "A Coupled-Adjoint Sensitivity Analysis Method for High-Fidelity Aero-Structural Design," *Optimization and Engineering*, Vol. 6, No. 1, March 2005, pp. 33–62.  
doi: [10.1023/B:OPTE.0000048536.47956.62](https://doi.org/10.1023/B:OPTE.0000048536.47956.62) cited on p. 38
- 37 Hwang, J. T. and Martins, J. R. R. A., "A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives," *ACM Transactions on Mathematical Software*, Vol. 44, No. 4, June 2018, Article 37.  
doi: [10.1145/3182393](https://doi.org/10.1145/3182393) cited on pp. 38, 207, 380
- 38 Wright, M. H., "The interior-point revolution in optimization: History, recent developments, and lasting consequences," *Bulletin of the American Mathematical Society*, Vol. 42, 2005, pp. 39–56.  
cited on p. 39
- 39 Grant, M., Boyd, S., and Ye, Y., "Global optimization—from theory to implementation," Liberti, L. and Maculan, N., Eds. Springer, 2006, ch. Disciplined Convex Programming, pp. 155–210.  
cited on p. 39
- 40 Wengert, R. E., "A Simple Automatic Derivative Evaluation Program," *Commun. ACM*, Vol. 7, No. 8, August 1964, pp. 463–464, ISSN: 0001-0782.  
doi: [10.1145/355586.364791](https://doi.org/10.1145/355586.364791) cited on p. 39
- 41 Speelpenning, B., "Compiling fast partial derivatives of functions given by algorithms," Ph.D. Dissertation, University of Illinois at Urbana–Champaign, January 1980.  
doi: [10.2172/5254402](https://doi.org/10.2172/5254402) cited on p. 39
- 42 Squire, W. and Trapp, G., "Using Complex Variables to Estimate Derivatives of Real Functions," *SIAM Review*, Vol. 40, No. 1, 1998, pp. 110–112, ISSN: 0036-1445 (print), 1095-7200 (electronic).  
cited on p. 39
- 43 Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The Complex Step Derivative Approximation," *ACM Transactions on Mathematical Software*, Vol. 29, No. 3, 2003, pp. 245–262, September.  
doi: [10.1145/838250.838251](https://doi.org/10.1145/838250.838251) cited on pp. 40, 185
- 44 Torczon, V., "On the Convergence of Pattern Search Algorithms," *SIAM Journal on Optimization*, Vol. 7, No. 1, February 1997, pp. 1–25.  
cited on p. 40
- 45 Jones, D., Perttunen, C., and Stuckman, B., "Lipschitzian optimization without the Lipschitz constant," *Journal of Optimization Theory and Application*, Vol. 79, No. 1, October 1993, pp. 157–181.  
cited on pp. 40, 221, 222

- 46 Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, 1983, pp. 671–680.  
doi: [10.1126/science.220.4598.671](https://doi.org/10.1126/science.220.4598.671) cited on p. 40
- 47 Kennedy, J. and Eberhart, R. C., "Particle Swarm Optimization," *IEEE International Conference on Neural Networks*, Vol. IV, Piscataway, NJ, 1995, pp. 1942–1948. cited on p. 40
- 48 Forrester, A. I. and Keane, A. J., "Recent advances in surrogate-based optimization," *Progress in Aerospace Sciences*, Vol. 45, No. 1, 2009, pp. 50–79, issn: 0376-0421. cited on p. 40  
doi: [10.1016/j.paerosci.2008.11.001](https://doi.org/10.1016/j.paerosci.2008.11.001)
- 49 Bottou, L., Curtis, F. E., and Nocedal, J., "Optimization Methods for Large-Scale Machine Learning," *SIAM Review*, Vol. 60, No. 2, 2018, pp. 223–311. cited on p. 41  
doi: [10.1137/16M1080173](https://doi.org/10.1137/16M1080173)
- 50 Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M., "Automatic Differentiation in Machine Learning: A Survey," *Journal of Machine Learning Research*, Vol. 18, No. 1, January 2018, pp. 5595–5637. cited on p. 41
- 51 Wilson, G., Aruliah, D. A., Brown, C. T., Hong, N. P. C., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., and Wilson, P., "Best Practices for Scientific Computing," *PLoS Biology*, Vol. 12, No. 1, 2014, e1001745. cited on p. 53  
doi: [10.1371/journal.pbio.1001745](https://doi.org/10.1371/journal.pbio.1001745)
- 52 Grotker, T., Holtmann, U., Keding, H., and Wloka, M., *The Developer's Guide to Debugging*, 2nd. 2012. cited on p. 54
- 53 Ascher, U. M. and Greif, C., *A first course in numerical methods*. SIAM, 2011. cited on p. 58
- 54 Saad, Y., *Iterative Methods for Sparse Linear Systems*, 2<sup>nd</sup>. SIAM, 2003. cited on p. 59
- 55 Hager, W. W. and Zhang, H., "A New Conjugate Gradient Method with Guaranteed Descent and an Efficient Line Search," *SIAM Journal on Optimization*, Vol. 16, No. 1, January 2005, pp. 170–192, issn: 1095-7189. cited on p. 87  
doi: [10.1137/030601880](https://doi.org/10.1137/030601880)
- 56 Nocedal, J. and Wright, S. J., *Numerical Optimization*, 2nd. Springer-Verlag, 2006. cited on pp. 91, 112, 113, 152, 158

- 57 Broyden, C. G., "The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations," *IMA Journal of Applied Mathematics*, Vol. 6, No. 1, 1970, pp. 76–90, issn: 1464-3634.  
doi: [10.1093/imamat/6.1.76](https://doi.org/10.1093/imamat/6.1.76) cited on p. 102
- 58 Fletcher, R., "A new approach to variable metric algorithms," *The Computer Journal*, Vol. 13, No. 3, March 1970, pp. 317–322, issn: 1460-2067.  
doi: [10.1093/comjnl/13.3.317](https://doi.org/10.1093/comjnl/13.3.317) cited on p. 102
- 59 Goldfarb, D., "A family of variable-metric methods derived by variational means," *Mathematics of Computation*, Vol. 24, No. 109, January 1970, pp. 23–23, issn: 0025-5718.  
doi: [10.1090/s0025-5718-1970-0258249-6](https://doi.org/10.1090/s0025-5718-1970-0258249-6) cited on p. 102
- 60 Shanno, D. F., "Conditioning of quasi-Newton methods for function minimization," *Mathematics of Computation*, Vol. 24, No. 111, September 1970, pp. 647–647, issn: 0025-5718.  
doi: [10.1090/s0025-5718-1970-0274029-x](https://doi.org/10.1090/s0025-5718-1970-0274029-x) cited on p. 102
- 61 Fletcher, R., *Practical Methods of Optimization*, 2nd. Wiley, 1987. cited on pp. 103, 153
- 62 Conn, A. R., Gould, N. I. M., and Toint, P. L., *Trust Region Methods*. SIAM, January 2000.  
isbn: [0898714605](https://doi.org/10.1137/14605) cited on pp. 111, 112, 113
- 63 Steihaug, T., "The Conjugate Gradient Method and Trust Regions in Large Scale Optimization," *SIAM Journal on Numerical Analysis*, Vol. 20, No. 3, June 1983, pp. 626–637, issn: 1095-7170.  
doi: [10.1137/0720042](https://doi.org/10.1137/0720042) cited on p. 112
- 64 Murray, W., "Analytical expressions for the eigenvalues and eigenvectors of the Hessian matrices of barrier and penalty functions," *Journal of Optimization Theory and Applications*, Vol. 7, No. 3, March 1971, pp. 189–196.  
doi: [10.1007/bf00932477](https://doi.org/10.1007/bf00932477) cited on p. 146
- 65 Forsgren, A., Gill, P. E., and Wright, M. H., "Interior Methods for Nonlinear Optimization," *SIAM Review*, Vol. 44, No. 4, January 2002, pp. 525–597.  
doi: [10.1137/s0036144502414942](https://doi.org/10.1137/s0036144502414942) cited on p. 146
- 66 Gill, P. E., Murray, W., and Saunders, M. A., "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM Review*, Vol. 47, No. 1, 2005, pp. 99–131.  
doi: [10.1137/S0036144504446096](https://doi.org/10.1137/S0036144504446096) cited on p. 152

- 67 Liu, D. C. and Nocedal, J., "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, Vol. 45, No. 1-3, August 1989, pp. 503–528.  
doi: [10.1007/bf01589116](https://doi.org/10.1007/bf01589116) cited on p. 153
- 68 Wächter, A. and Biegler, L. T., "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, Vol. 106, No. 1, April 2005, pp. 25–57.  
doi: [10.1007/s10107-004-0559-y](https://doi.org/10.1007/s10107-004-0559-y) cited on p. 157
- 69 Byrd, R. H., Hribar, M. E., and Nocedal, J., "An Interior Point Algorithm for Large-Scale Nonlinear Programming," *SIAM Journal on Optimization*, Vol. 9, No. 4, January 1999, pp. 877–900.  
doi: [10.1137/s1052623497325107](https://doi.org/10.1137/s1052623497325107) cited on p. 157
- 70 Fletcher, R. and Leyffer, S., "Nonlinear programming without a penalty function," *Mathematical Programming*, Vol. 91, No. 2, January 2002, pp. 239–269.  
doi: [10.1007/s101070100244](https://doi.org/10.1007/s101070100244) cited on p. 160
- 71 Fletcher, R., Leyffer, S., and Toint, P., "A Brief History of Filter Methods," ANL/MCS-P1372-0906, Argonne National Laboratory, September 2006. cited on pp. 160, 161
- 72 Benson, H. Y., Vanderbei, R. J., and Shanno, D. F., "Interior-Point Methods for Nonconvex Nonlinear Programming: Filter Methods and Merit Functions," *Computational Optimization and Applications*, Vol. 23, No. 2, 2002, pp. 257–272.  
doi: [10.1023/a:1020533003783](https://doi.org/10.1023/a:1020533003783) cited on p. 160
- 73 Kreisselmeier, G. and Steinhauser, R., "Systematic Control Design by Optimizing a Vector Performance Index," *IFAC Proceedings Volumes*, Vol. 12, No. 7, September 1979, pp. 113–117, ISSN: 1474-6670.  
doi: [10.1016/s1474-6670\(17\)65584-8](https://doi.org/10.1016/s1474-6670(17)65584-8) cited on p. 163
- 74 Hoerner, S. F., *Fluid-Dynamic Drag*. 1965. cited on p. 169
- 75 Lyness, J. N., "Numerical Algorithms Based on the Theory of Complex Variable," *Proceedings — ACM National Meeting*, Washington DC: Thompson Book Co., 1967, pp. 125–133. cited on p. 181
- 76 Lyness, J. N. and Moler, C. B., "Numerical Differentiation of Analytic Functions," *SIAM Journal on Numerical Analysis*, Vol. 4, No. 2, 1967, pp. 202–210, ISSN: 0036-1429 (print), 1095-7170 (electronic). cited on p. 181

- 77 Lantoine, G., Russell, R. P., and Dargent, T., “[Using Multicomplex Variables for Automatic Computation of High-Order Derivatives](#),” *ACM Transactions on Mathematical Software*, Vol. 38, No. 3, April 2012, pp. 1–21, issn: 0098-3500.  
doi: [10.1145/2168773.2168774](https://doi.org/10.1145/2168773.2168774) cited on p. 182
- 78 Fike, J. and Alonso, J., “[The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations](#),” *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, January 2011  
doi: [10.2514/6.2011-886](https://doi.org/10.2514/6.2011-886) cited on p. 182
- 79 Griewank, A., *Evaluating Derivatives*. Philadelphia: SIAM, 2000. cited on p. 186
- 80 Naumann, U., “[The art of differentiating computer programs—an introduction to algorithmic differentiation](#).” SIAM, 2011. cited on p. 186
- 81 Hascoët, L. and Pascual, V., “[TAPENADE 2.1 User’s Guide](#),” Technical report 300, INRIA, 2004. cited on p. 193
- 82 Griewank, A., Juedes, D., and Utke, J., “[Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++](#),” *ACM Transactions on Mathematical Software*, Vol. 22, No. 2, June 1996, pp. 131–167, issn: 0098-3500.  
doi: [10.1145/229473.229474](https://doi.org/10.1145/229473.229474) cited on p. 193
- 83 Wiltschko, A. B., Merriënboer, B. van, and Moldovan, D., “[Tangent: Automatic differentiation using source code transformation in Python](#),” 2017. cited on p. 193
- 84 Revels, J., Lubin, M., and Papamarkou, T., “[Forward-Mode Automatic Differentiation in Julia](#),” arXiv:1607.07892, July 2016. cited on p. 193
- 85 Neidinger, R. D., “[Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming](#),” *SIAM Review*, Vol. 52, No. 3, January 2010, pp. 545–563.  
doi: [10.1137/080743627](https://doi.org/10.1137/080743627) cited on p. 193
- 86 Betancourt, M., “[A geometric theory of higher-order automatic differentiation](#),” arXiv:1812.11592 [stat.CO], December 2018. cited on p. 193
- 87 Curtis, A. R., Powell, M. J. D., and Reid, J. K., “[On the Estimation of Sparse Jacobian Matrices](#),” *IMA Journal of Applied Mathematics*, Vol. 13, No. 1, February 1974, pp. 117–119, issn: 1464-3634.  
doi: [10.1093/imamat/13.1.117](https://doi.org/10.1093/imamat/13.1.117) cited on p. 202
- 88 Gebremedhin, A. H., Manne, F., and Pothen, A., “[What Color Is Your Jacobian? Graph Coloring for Computing Derivatives](#),” *SIAM Review*, Vol. 47, No. 4, January 2005, pp. 629–705, issn: 1095-7200.  
doi: [10.1137/s0036144504444711](https://doi.org/10.1137/s0036144504444711) cited on p. 203

- 89 Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, April 2019, pp. 1075–1104.  
doi: [10.1007/s00158-019-02211-z](https://doi.org/10.1007/s00158-019-02211-z) cited on pp. 203, 207, 382
- 90 Ning, A., "Using Blade Element Momentum Methods with Gradient-Based Design Optimization," 2020, (in review). cited on p. 204
- 91 Martins, J. R. R. A. and Hwang, J. T., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models," *AIAA Journal*, Vol. 51, No. 11, November 2013, pp. 2582–2599.  
doi: [10.2514/1.J052184](https://doi.org/10.2514/1.J052184) cited on p. 204
- 92 Yu, Y., Lyu, Z., Xu, Z., and Martins, J. R. R. A., "On the Influence of Optimization Algorithm and Starting Design on Wing Aerodynamic Shape Optimization," *Aerospace Science and Technology*, Vol. 75, April 2018, pp. 183–199.  
doi: [10.1016/j.ast.2018.01.016](https://doi.org/10.1016/j.ast.2018.01.016) cited on p. 213
- 93 Rios, L. M. and Sahinidis, N. V., "Derivative-free optimization: A review of algorithms and comparison of software implementations," *Journal of Global Optimization*, Vol. 56, 2013, pp. 1247–1293.  
doi: [10.1007/s10898-012-9951-y](https://doi.org/10.1007/s10898-012-9951-y) cited on pp. 213, 214
- 94 Conn, A. R., Scheinberg, K., and Vicente, L. N., *Introduction to Derivative-Free Optimization*. SIAM, 2009. cited on p. 215
- 95 Audet, C. and Hare, W., *Derivative-Free and Blackbox Optimization*. Springer, 2017.  
doi: [10.1007/978-3-319-68913-5](https://doi.org/10.1007/978-3-319-68913-5) cited on p. 215
- 96 Le Digabel, S., "Algorithm 909: NOMAD: Nonlinear Optimization with the MADS algorithm," *ACM Transactions on Mathematical Software*, Vol. 37, No. 4, 2011, pp. 1–15. cited on p. 215
- 97 Jones, D. R., "Direct global optimization algorithm," *Encyclopedia of Optimization*, Floudas, C. A. and Pardalos, P. M., Eds. Boston, MA: Springer US, 2009, pp. 725–735, ISBN: 978-0-387-74759-0  
. doi: [10.1007/978-0-387-74759-0\\_128](https://doi.org/10.1007/978-0-387-74759-0_128) cited on pp. 215, 227
- 98 Simon, D., *Evolutionary Optimization Algorithms*. John Wiley & Sons, June 2013.  
ISBN: [1118659503](https://doi.org/10.1007/978-0-387-74759-0_128) cited on pp. 216, 236
- 99 Barricelli, N., "Esempi numerici di processi di evoluzione," *Methodos*, 1954, pp. 45–68. cited on p. 230

- 100 Jong, K. A. D., "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, 1975. cited on p. 230
- 101 Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T., "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, Vol. 6, No. 2, April 2002, pp. 182–197. doi: 10.1109/4235.996017 cited on pp. 232, 285
- 102 Deb, K., *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, July 2001. ISBN: 047187339X cited on p. 238
- 103 Eberhart, R. and Kennedy, J. A., "New Optimizer Using Particle Swarm Theory," *Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 1995, pp. 39–43. cited on p. 240
- 104 Gutin, G., Yeo, A., and Zverovich, A., "Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the TSP," *Discrete Applied Mathematics*, Vol. 117, No. 1-3, March 2002, pp. 81–86, ISSN: 0166-218X. doi: 10.1016/s0166-218x(01)00195-0 cited on p. 260
- 105 Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, May 1983, pp. 671–680, ISSN: 1095-9203. doi: 10.1126/science.220.4598.671 cited on p. 269
- 106 Černý, V., "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, Vol. 45, No. 1, January 1985, pp. 41–51, ISSN: 1573-2878. doi: 10.1007/bf00940812 cited on p. 269
- 107 Andresen, B. and Gordon, J. M., "Constant thermodynamic speed for minimizing entropy production in thermodynamic processes and simulated annealing," *Physical Review E*, Vol. 50, No. 6, December 1994, pp. 4346–4351, ISSN: 1095-3787. doi: 10.1103/physreve.50.4346 cited on p. 270
- 108 Lin, S., "Computer Solutions of the Traveling Salesman Problem," *Bell System Technical Journal*, Vol. 44, No. 10, December 1965, pp. 2245–2269, ISSN: 0005-8580. doi: 10.1002/j.1538-7305.1965.tb04146.x cited on p. 271

- 109 Press, W. H., Wevers, J., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., and Vetterling, W. T., *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, October 1992.  
ISBN: 0521431085 cited on p. 272
- 110 Haimes, Y. Y., Lasdon, L. S., and Wismer, D. A., "On a Bicriterion Formulation of the Problems of Integrated System Identification and System Optimization," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-1, No. 3, July 1971, pp. 296–297.  
doi: 10.1109/tsmc.1971.4308298 cited on p. 282
- 111 Das, I. and Dennis, J. E., "Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems," *SIAM Journal on Optimization*, Vol. 8, No. 3, August 1998, pp. 631–657.  
doi: 10.1137/s1052623496307510 cited on p. 282
- 112 Ismail-Yahaya, A. and Messac, A., "Effective generation of the Pareto frontier using the Normal Constraint method," *40th AIAA Aerospace Sciences Meeting & Exhibit*, American Institute of Aeronautics and Astronautics, January 2002.  
doi: 10.2514/6.2002-178 cited on p. 285
- 113 Messac, A. and Mattson, C. A., "Normal Constraint Method with Guarantee of Even Representation of Complete Pareto Frontier," *AIAA Journal*, Vol. 42, No. 10, October 2004, pp. 2101–2111.  
doi: 10.2514/1.8977 cited on p. 285
- 114 Hancock, B. J. and Mattson, C. A., "The smart normal constraint method for directly generating a smart Pareto set," *Structural and Multidisciplinary Optimization*, Vol. 48, No. 4, June 2013, pp. 763–775.  
doi: 10.1007/s00158-013-0925-6 cited on p. 285
- 115 Schaffer, J. D., "Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms." Ph.D. Dissertation, Vanderbilt University, Nashville, TN, 1984. cited on p. 285
- 116 Deb, K., *Introduction to evolutionary multiobjective optimization, Multiobjective Optimization*, Springer Berlin Heidelberg, 2008, pp. 59–96.  
doi: 10.1007/978-3-540-88908-3\_3 cited on p. 285
- 117 Kung, H. T., Luccio, F., and Preparata, F. P., "On Finding the Maxima of a Set of Vectors," *Journal of the ACM*, Vol. 22, No. 4, October 1975, pp. 469–476.  
doi: 10.1145/321906.321910 cited on p. 286

- 118 Forrester, A., Sobester, A., and Keane, A., *Engineering Design via Surrogate Modelling: A Practical Guide*. John Wiley & Sons, September 2008.  
ISBN: 0470770791      cited on p. 295
- 119 Rajnarayan, D., Haas, A., and Kroo, I., "A Multifidelity Gradient-Free Optimization Method and Application to Aerodynamic Design," *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, September 2008  
doi: 10.2514/6.2008-6020      cited on p. 305
- 120 Ruder, S., "An overview of gradient descent optimization algorithms," arXiv:1609.04747, 2016.      cited on p. 312
- 121 Goh, G., "Why Momentum Really Works," *Distill*, 2017.      cited on p. 312  
doi: 10.23915/distill.00006
- 122 Diamond, S. and Boyd, S., "Convex Optimization with Abstract Linear Operators," *2015 IEEE International Conference on Computer Vision (ICCV)*, IEEE, December 2015.  
doi: 10.1109/iccv.2015.84      cited on p. 316
- 123 Boyd, S. P. and Vandenberghe, L., *Convex Optimization*. Cambridge University Press, March 2004.  
ISBN: 0521833787      cited on p. 318
- 124 Lobo, M. S., Vandenberghe, L., Boyd, S., and Lebret, H., "Applications of second-order cone programming," *Linear Algebra and its Applications*, Vol. 284, No. 1-3, November 1998, pp. 193–228.  
doi: 10.1016/s0024-3795(98)10032-0      cited on p. 318
- 125 Vandenberghe, L. and Boyd, S., "Applications of semidefinite programming," *Applied Numerical Mathematics*, Vol. 29, No. 3, March 1999, pp. 283–299.  
doi: 10.1016/s0168-9274(98)00098-1      cited on p. 318
- 126 Vandenberghe, L. and Boyd, S., "Semidefinite Programming," *SIAM Review*, Vol. 38, No. 1, March 1996, pp. 49–95.  
doi: 10.1137/1038003      cited on p. 318
- 127 Parikh, N. and Boyd, S., "Block splitting for distributed optimization," *Mathematical Programming Computation*, Vol. 6, No. 1, October 2013, pp. 77–102.  
doi: 10.1007/s12532-013-0061-8      cited on p. 318
- 128 Grant, M., Boyd, S., and Ye, Y., *Disciplined convex programming*, *Global Optimization*, Kluwer Academic Publishers, 2006, pp. 155–210.  
doi: 10.1007/0-387-30528-9\_7      cited on p. 324

- 129 Hoburg, W. and Abbeel, P., "Geometric Programming for Aircraft Design Optimization," *AIAA Journal*, Vol. 52, No. 11, November 2014, pp. 2414–2426.  
doi: [10.2514/1.j052732](https://doi.org/10.2514/1.j052732) cited on p. 327
- 130 Hoburg, W., Kirschen, P., and Abbeel, P., "Data fitting with geometric-programming-compatible softmax functions," *Optimization and Engineering*, Vol. 17, No. 4, August 2016, pp. 897–918.  
doi: [10.1007/s11081-016-9332-3](https://doi.org/10.1007/s11081-016-9332-3) cited on p. 327
- 131 Kirschen, P. G., York, M. A., Ozturk, B., and Hoburg, W. W., "Application of Signomial Programming to Aircraft Design," *Journal of Aircraft*, Vol. 55, No. 3, May 2018, pp. 965–987.  
doi: [10.2514/1.c034378](https://doi.org/10.2514/1.c034378) cited on p. 328
- 132 York, M. A., Hoburg, W. W., and Drela, M., "Turbofan Engine Sizing and Tradeoff Analysis via Signomial Programming," *Journal of Aircraft*, Vol. 55, No. 3, May 2018, pp. 988–1003.  
doi: [10.2514/1.c034463](https://doi.org/10.2514/1.c034463) cited on p. 328
- 133 Nemec, M., Zingg, D. W., and Pulliam, T. H., "Multipoint and Multi-Objective Aerodynamic Shape Optimization," *AIAA Journal*, Vol. 42, No. 6, June 2004, pp. 1057–1065. cited on p. 337
- 134 Smolyak, S. A., "Quadrature and interpolation formulas for tensor products of certain classes of functions," *Dokl. Akad. Nauk SSSR*, Vol. 148, No. 5, 1963, pp. 1042–1045. cited on p. 344
- 135 Smith, R. C., *Uncertainty Quantification: Theory, Implementation, and Applications*. SIAM, December 2013. cited on p. 347  
ISBN: [1611973228](https://doi.org/10.111973228)
- 136 Cacuci, D., *Sensitivity & Uncertainty Analysis, Volume 1*. Chapman and Hall/CRC, May 2003. cited on p. 347  
doi: [10.1201/9780203498798](https://doi.org/10.1201/9780203498798)
- 137 Parkinson, A., Sorensen, C., and Pourhassan, N., "A General Approach for Robust Optimal Design," *Journal of Mechanical Design*, Vol. 115, No. 1, 1993, p. 74. cited on p. 348  
doi: [10.1115/1.2919328](https://doi.org/10.1115/1.2919328)
- 138 Wiener, N., "The Homogeneous Chaos," *American Journal of Mathematics*, Vol. 60, No. 4, October 1938, p. 897. cited on pp. 348, 349  
doi: [10.2307/2371268](https://doi.org/10.2307/2371268)

- 139 Eldred, M., Webster, C., and Constantine, P., "Evaluation of Non-Intrusive Approaches for Wiener-Askey Generalized Polynomial Chaos," *49th AIAA Structures, Structural Dynamics, and Materials Conference*, American Institute of Aeronautics and Astronautics, April 2008.  
doi: [10.2514/6.2008-1892](https://doi.org/10.2514/6.2008-1892) cited on p. 352
- 140 Adams, B., Bauman, L., Bohnhoff, W., Dalbey, K., Ebeida, M., Eddy, J., Eldred, M., Hough, P., Hu, K., Jakeman, J., Stephens, J., Swiler, L., Vigil, D., and Wildey, T., "Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.0 User's Manual," Sandia Technical Report SAND2014-4633, Sandia National Laboratories, November 2015.  
cited on p. 354
- 141 Feinberg, J. and Langtangen, H. P., "Chaospy: An open source tool for designing methods of uncertainty quantification," *Journal of Computational Science*, Vol. 11, November 2015, pp. 46–57.  
doi: [10.1016/j.jocs.2015.08.008](https://doi.org/10.1016/j.jocs.2015.08.008) cited on p. 354
- 142 Kroo, I. M., *MDO for large-scale design, Multidisciplinary Design Optimization: State-of-the-Art*, Alexandrov, N. and Hussaini, M. Y., Eds., SIAM, 1997, pp. 22–44.  
cited on p. 359
- 143 Biegler, L. T., Ghattas, O., Heinkenschloss, M., and Bloemen Waanders, B. van, Eds., *Large-Scale PDE-Constrained Optimization*. Springer-Verlag, 2003.  
cited on p. 379
- 144 Braun, R. D., "Collaborative Optimization: An Architecture for Large-Scale Distributed Design," Ph.D. Dissertation, Stanford University, Stanford, CA 94305, 1996.  
cited on p. 385
- 145 Tedford, N. P. and Martins, J. R. R. A., "Benchmarking Multi-disciplinary Design Optimization Algorithms," *Optimization and Engineering*, Vol. 11, No. 1, February 2010, pp. 159–183.  
doi: [10.1007/s11081-009-9082-6](https://doi.org/10.1007/s11081-009-9082-6) cited on p. 394
- 146 Golovidov, O., Kodiyalam, S., Marineau, P., Wang, L., and Rohl, P., "Flexible implementation of approximation concepts in an MDO framework," *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, American Institute of Aeronautics and Astronautics, 1998.  
doi: [10.2514/6.1998-4959](https://doi.org/10.2514/6.1998-4959) cited on p. 395
- 147 Balabanov, V., Charpentier, C., Ghosh, D. K., Quinn, G., Vanderplaats, G., and Venter, G., "VisualDOC: A Software System for General Purpose Integration and Design Optimization," *9th* cited on p. 395

*AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, GA, 2002.*

- 148 Barnes, G. K., "A Comparative Study of Nonlinear Optimization Codes," Master's thesis, The University of Texas at Austin, 1967. cited on p. 419
- 149 Venkayya, V., "Design of optimum structures," *Computers & Structures*, Vol. 1, No. 1-2, August 1971, pp. 265–309, ISSN: 0045-7949. cited on p. 420  
doi: [10.1016/0045-7949\(71\)90013-7](https://doi.org/10.1016/0045-7949(71)90013-7)

## Index

---

- aerostructural analysis, 361
- asymptotic convergence, 55
- back propagation, 310
- bean function, 97, 414
- binary design varialbes, 249
- black-box model, 17
- Boltzman distribution, 269
- brachistochrone problem, 33, 121
- branch and bound, 252
- callback functions, 14
- constraints
  - active, 12
  - equality, 11
  - inactive, 12
  - inequality, 12
  - infeasible, 12
- convergence
  - rate of, 54
- convex function, 19, 317
- cross validation, 301
- cumulative density function, 335
- curse of dimensionality, 295, 344
- debugging, 53
- design variables, 7
  - binary, 249
  - converting integer to binary, 252
  - discrete, 249
  - integer, 249
- disciplined convex programming, 317, 324
- discrete design varialbes, 249
- dominated, 161, 280
- dynamic programming principle of optimality, 263
- elitism, 240, 289
- error
  - absolute, 48
  - relative, 48
  - roundoff, 48
- errors
  - programming, 53
- exhaustive search, 251
- expected improvement, 304
- explicit function, 45
- finite-difference method, 174
- fixed-point iterations, 59
- forward propagation, 343
- genetic algorithm
  - binary, 273
- geometric program, 325
- greedy algorithms, 259
- Hartmann function, 414
- implicity equations, 45
- infill, 304
- integer design varialbes, 249
- inversion sampling, 297
- iterative solver, 52, 54, 59
- Jones function, 230, 414
- knapsack problem, 265
- Kriging, 298

kriging, 303  
Krylov subspace methods, 59  
Latin hypercube sampling, 296, 345  
least squares, 299, 321  
linear  
    convergence, 55  
linear mixed integer programming, 253  
linear program, 318  
linear regression model, 298  
  
Markov chain, 261  
maximization as minimization, 10  
mean, 334  
memoization, 262  
minibatch, 311  
minimum  
    global, local, weak, 18  
Monte Carlo, 344  
multidisciplinary design optimization, 2  
multimodal function, 18  
  
neural networks, 306  
Newton  
    Isaac, 33  
    method for minimization, 100  
    method for root finding, 59  
NP-complete, 250  
numerical noise, 52  
  
objective function, 9  
    multiple (multiobjective), 277  
optimization problem statement, 14  
optimization software, 14  
optimization under uncertainty, 332  
overfitting, 301  
  
Pareto  
    anchor points, 282  
    optimal, 280  
    set, 280  
    utopia point, 283  
polynomial chaos, 348  
probability density function, 335  
  
quadratic  
    convergence, 55  
quadratic program, 320  
quadratically-constrained quadratic program, 112, 323  
quadrature, 343  
  
reliable design, 332, 341  
residual, 45  
residual form, 45  
robust design, 332, 337  
robustness  
    design, 332  
  
simulated annealing, 269  
state variables, 45  
successive over-relaxation (SOR), 59  
superlinear convergence, 56  
surrogate-based optimization, 295  
  
tabulation, 262  
Taylor's series, 347  
training data, 295  
traveling salesman, 250, 271  
  
unimodal function, 18  
  
variance, 334  
  
warm start, 121, 409  
weighted directed graph, 259

