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> or <aning@byu.edu>.

Draft compiled on Monday 13th April, 2020 at 22:06

Publication
First electronic edition Jan 2020.

Contents

Preface

In spite of its usefulness, design optimization remains underused in industry. There are several reasons for this, one of which 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 multidisciplinary design optimization being increasingly used in industry, including the major aircraft manufacturers.

While "multidisciplinary design optimization" is often associated with certain methods and architectures, our use of this term as the title of this book has a broader meaning. The design aspect reflects our focus on practical engineering usage. Most engineering optimization problems are design oriented, whether the design aspect manifests itself in a physical device or not. These problems almost always have constraints, are highly nonlinear, and often have large numbers of heterogeneous design variables with varying scales. The word "multidisciplinary" reflects the complex nature of modern engineering problems that require computationally expensive simulations. We also discuss techniques unique to multidisciplinary problems, such as hierarchical solvers and coupled derivatives.

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 we do review many of the important concepts on these topics along the way. 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 special cases. Chapters 1–6 dive deep into fundamentals, whereas Chapters 7–12 provide broader overviews of more specialized or advanced topics. An undergraduate course

would likely cover the fundamentals (perhaps with the exception of Chapter 2), whereas a typical graduate course would cover the fundamentals plus a selected subset of the remaining advanced topics.

Our philosophy in the exposition is to provide a detailed enough explanation and analysis of optimization techniques so that the reader can implement a basic working version. While we do not encourage readers to use their own implementations in general, this is a useful exercise to truly understand the given method. A deeper understanding of these techniques is useful not only for developers, but also 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 technique, and how to use it more effectively.

Acknowledgments

We wish to thank Aaron Lu for translating many of our figures to a readable and precise format. We are indebted to many students at our respective institutions that provided feedback on concepts, examples, and drafts of the manuscript.

Joaquim Martins and Andrew Ning

CHAPTER 1

Introduction

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 the species. Birds optimize the shape of their wings in real time, and even 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.

Optimization is often used to mean improvement, but mathematically it is a much more precise concept: it means 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 we can all identify with a desire to make things better.

While some simple optimization problems can be solved analytically, 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.

Most systems rarely work in isolation and are linked to other systems, so they are *multidisciplinary* by default. This gave rise to the field of multidisciplinary design optimization (MDO). MDO is the application of numerical optimization techniques to the design of engineering systems that involve multiple disciplines. MDO emerged in the 1980s following the success of the application of numerical optimization techniques to structural design in the 1970s. Aircraft design was one of the first applications of MDO because there is much to be gained by the simultaneous consideration of the various disciplines involved (such as structures, aerodynamics, propulsion, stability, and controls), which are tightly coupled. In this book, we replace "discipline" with the broader

concept of *component*, in part because not all engineering systems can be neatly divided into disciplines.

There are two main benefits when considering multiple disciplines. One is that the full coupled system is modeled taking the interactions within the system into account. That way, the effect of a change in a given component is propagated to the system level, giving us the true final effect that accounts for all the couplings. The other benefit is that because MDO optimizes all the components simultaneously, it achieves a better result than when optimizing one component at the time.

Before seeking to understand the multidisciplinary aspects of MDO, we first need to understand design optimization. This book provides an introduction to a variety of numerical techniques that are useful in engineering design optimization. This book spans a broad spectrum of optimization algorithms because no algorithm is the best for solving all problems. We cannot hope to explain all existing optimization algorithms and techniques, so we restrict ourselves to a selection of techniques that represents the current state-of-the-art.

In the rest of this chapter, we introduce the design optimization process and how the corresponding optimization problem is formulated. When explaining the components of an optimization problem, we take the opportunity to discuss the classification of these problems. We then discuss how optimization algorithms can be classified. These two classifications are essential when deciding which algorithm to use to solve a given optimization problem.

1.1 Design Optimization Process

The design optimization process follows a similar iterative procedure to the conventional design process, with a few key differences, as highlighted in Fig. 1.1. Both approaches must have a starting point, which is typically a baseline design derived from past designs, intuition, or an initial idea. In the conventional design process, this baseline design is analyzed in some way to determine its performance. This could involve numerical modeling or actual building and testing. The design is then evaluated, and based on the results, the designer decides whether the design is good enough or not. If the answer is no—which is likely to be the case for at least the first few iterations—the designer will change the design based on intuition, experience, or trade studies. When the design is satisfactory, the designer will arrive at the final design.

This is an oversimplification of the true design process, because design in industry usually involves various stages, each with its own design process loop. In one design paradigm, the stages are conceptual, preliminary, and detailed design.

The design optimization process can be pictured using the same flow diagram, with modifications to some of the blocks, as highlighted on the right of Fig. 1.1. The evaluation of the design is strictly based on numerical values

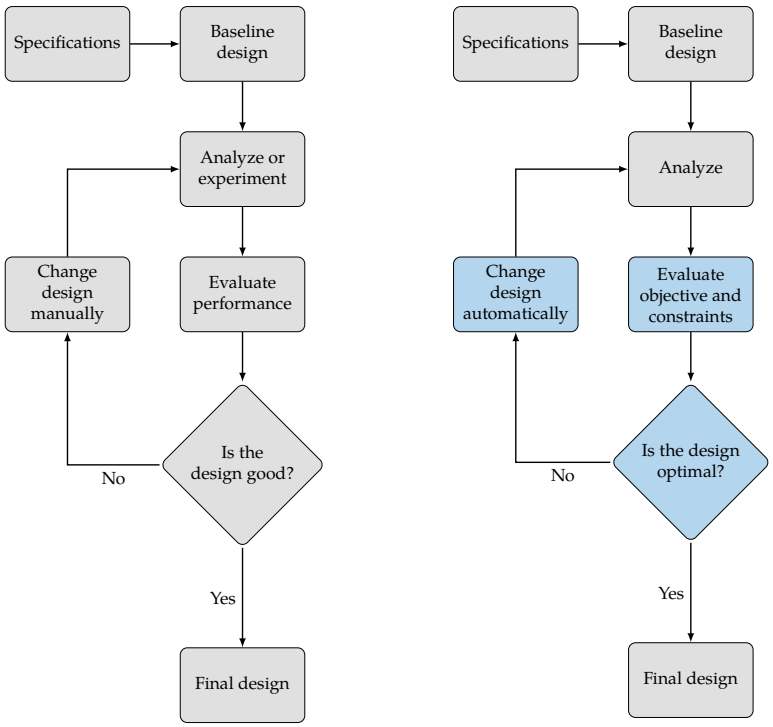


Figure 1.1: Conventional (left) versus design optimization process (right).

for the objective to be minimized and the constraints that need to be satisfied. When a rigorous optimization algorithm is used, the decision to finalize the design is made only when the current design satisfies the optimality conditions that ensure that no other design "close by" is better. The changes in the design are made automatically by the optimization algorithm and do not require intervention from the designer. This is not to suggest that this is a "push-button" process that does not require design expertise; on the contrary, more expertise is required. The designer must decide in advance what the objective is, which parameters can be changed, and what constraints are important. These decisions have profound effects on the outcome, so it is crucial that the designer formulate the optimization problem well.

In this book we, will tend to frame problems and discussions in the context of engineering design. However, you should keep in mind that the optimization methods are general and are used in other applications that may not be called design problems (e.g., optimal control, machine learning, and regression). In other words, we mean "design" in a general sense, where parameters are allowed to change toward improving an objective. We now discuss the components of the optimization problem formulation in more detail.

1.2 Optimization Problem Formulation

The optimization problem formulation deals with translating the designer's intent to a mathematical statement that can then be solved by an optimization algorithm. The optimization problem statement can be formally written as:

$$\begin{aligned} &\text{minimize} && f(x) && \\ &\text{with respect to} && x_i && i = 1, \ldots, n_x \\ &\text{subject to} && g_j(x) \leq 0 && j = 1, \ldots, n_g \\ & && h_k(x) = 0 && k = 1, \ldots, n_h \\ & && \underline{x}_i \leq x_i \leq \overline{x}_i && i = 1, \ldots, n_x \end{aligned} \tag{1.1}$$

where f is the *objective function*, x is a vector of n_x *design variables*, g is a vector of n_g *inequality constraints*, h is a vector of n_h *equality constraints*, and $\underline{x}$ and $\overline{x}$ are lower and upper bounds on the design variables. The values of the objective and constraint functions for a given set of design variables are computed using numerical models, which we elaborate on in Chapter 2.

Understanding how to formulate the optimization problem is important because a bad formulation will cause the optimization to fail or simply return a mathematical optimum that is not the engineering optimum—the proverbial "right answer to the wrong question". One important law of design optimization is that *the optimizer will exploit any weaknesses you might have in your formulation or model*. In the remainder of this section, we explain the components of the problem formulation in more detail.

To choose the most appropriate optimization algorithm to solve a given optimization problem, we must be able to classify the optimization problem 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 the solution of all types of problems. As we introduce the components of the problem formulation, we also classify the different types of problems that arise from the combination of components with different characteristics.

We classify optimization problems based on two main aspects: the problem formulation (objective, constraints, and design variables) and the characteristics of the objective and constraint functions, as shown in Fig. 1.2. We discuss the problem formulation aspect in this section, followed by a discussion of the function characteristics in Section 1.3.

1.2.1 Objective Function

The goal of a design optimization is to minimize or maximize a given performance measure. 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.

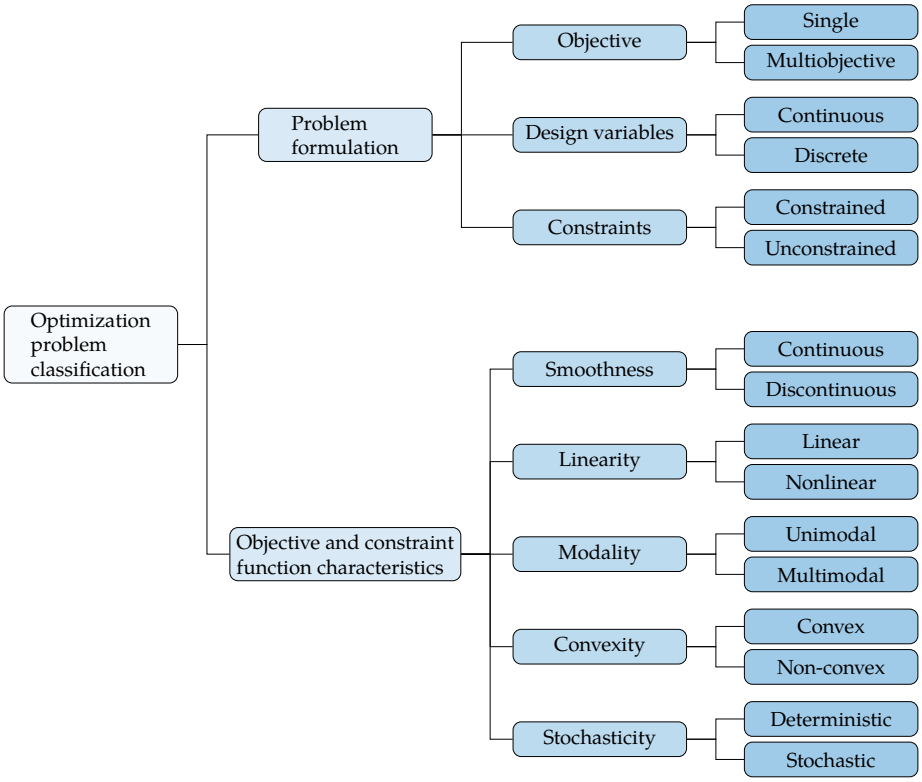


Figure 1.2: Optimization problems can be classified by attributes associated with the different aspects of the problem. The two main aspects are the problem formulation and the objective and constraint function characteristics.

The convention adopted in this book is that the objective function, f, is to be *minimized*. This convention does not prevent us from maximizing a function using the formal problem statement (1.1), since we can simply negate the function $(-f)$ or invert it $(1/f)$ to convert to a minimization problem. Negating is usually preferred to inverting because the latter changes the scaling of the problem and might cause a divide-by-zero problem.

To perform numerical optimization, we require the objective function to be computable as a function of the design variables (x). 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 2)

The choice of objective function is crucial. 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.

In some cases, choosing a single objective that represents the real world optimality is challenging or even impossible, and *multiobjective* optimization might be warranted. In vehicle design, for example, there is a tradeoff between acquisition cost and operating cost that is difficult to quantify. The challenge in such a case is to decide how much emphasis to put on one objective versus the another. Multiobjective optimization does not yield a single design, but rather a range of designs that settle for different tradeoffs between the objectives. This topic is discussed further in Chapter 7.

1.2.2 Design Variables

The design variables, represented by the vector x, are the parameters that the optimization algorithm is free to vary to minimize the objective function while satisfying the constraints. Design variables must be truly independent variables. This means that in the analysis of a given design, they must be input parameters that remain fixed in the analysis cycle. Furthermore, design variables must not depend on each other or any other parameter that is varying.

The optimization problem formulation allows us to limit the allowable range of the design variables by defining upper and lower *bounds* for each design variable (also known as *side constraints*). We distinguish these bounds from constraints, since they require a different numerical treatment when solving an optimization problem. Some optimization algorithms require upper and lower bounds for all variables, while others do not require any bounds at all. In either case, 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, the reasoning for that bound should be revisited. This is because designers sometimes set bounds that unnecessarily limit the optimization from obtaining a better objective.

Design variables can be either continuous or discrete. The continuous case corresponds to design variables that are real numbers that are allowed to vary continuously within a specified range. Most of this book focuses on algorithms that assume continuous design variables. The discrete case occurs when only a discrete set of design variables are allowed, which can be either real or integer. An example of discrete design variables is structural sizing where only components of certain thicknesses or cross-sectional areas are available. Integer design variables are a special case of discrete variables where the values are integer, like for example, the number of wheels in a vehicle. Optimization algorithms that handle discrete variables are discussed in Chapter 10.

1.2.3 Constraints

The vast majority of practical design optimization problems enforce 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.

When we restrict a function to being equal to a fixed quantity, 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 quantity, we have an *inequality constraint*, denoted by $g(x) \leq 0$. While we use "less or equal" by convention, you should be aware that some other texts 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. This conversion is especially important when using optimization software because you must convert the constraints to follow the convention used by that software. 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 $g(x) \leq 0$. A strict inequality, such as $g(x) < 0$, is never used because in that case, x can be arbitrarily close to the equality and the difference is not meaningful when using finite-precision arithmetic (which is always the case when using a computer).

Inequality constraints can be *active* or *inactive* at the optimum point. If inactive, then the corresponding constraint could have been left out of the problem with no change in its solution, as illustrated in Fig. 1.3. 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.

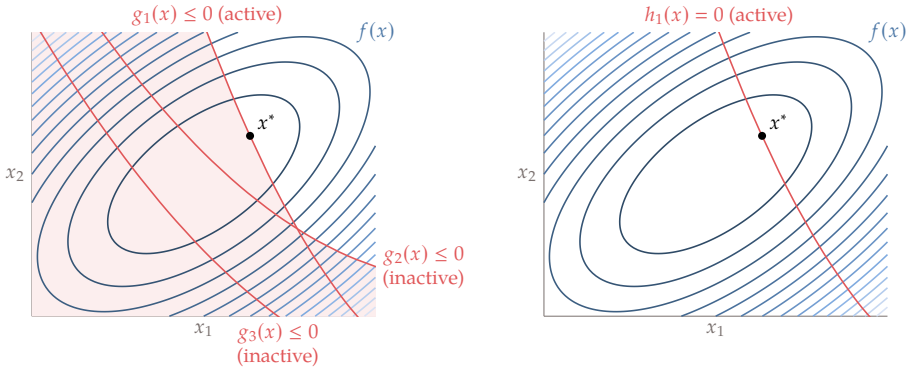


Figure 1.3: Example of two-dimensional problem with one active and two inactive inequality constraints (left). The red highlighted area indicates regions that are *infeasible* (i.e., the constraints are violated). This formulation is equivalent to a problem with a single equality constraint (right).

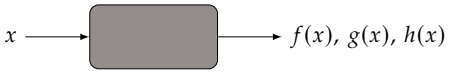


Figure 1.4: A model is considered a black box when we only see its inputs and outputs.

One common issue in optimization problem formulation is distinguishing objectives from constraints. For example, one might be tempted to maximize safety, but this would inevitably result in suboptimal performance (or no performance). Instead, we probably want maximum performance with sufficient safety, which can be enforced by an inequality constraint.

1.3 Function Characteristics

The characteristics of the objective and constraint functions 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. 1.4.

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 2, 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. 1.2. 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 examining the functions themselves. However, for the purposes of this discussion, we assume that the outputs of the black box 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 its derivatives. When the value of the function varies continuously, the function is said to be C^0 continuous. If in addition 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 greatly affects what type of optimization algorithm can be used. Figure 1.5 shows one-dimensional examples for these two possibilities and a function that is discontinuous.

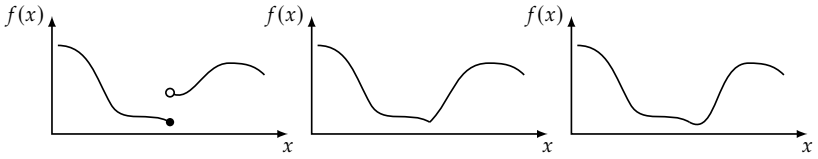


Figure 1.5: Discontinuous function (left), C^0 continuous function (center), and C^1 continuous function (right).

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, these algorithms still work with functions that have only 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*[1]. These problems are much easier to solve than general nonlinear ones and there are entire books and courses dedicated to the subject. An overview of linear programming is contained in Section 9.2.1. While there are a large number of problems that can be formulated as linear problems, most of the problems in engineering design 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 this.

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*.

[1]Historically, optimization problems were referred to as programming problems, so these types of problems were (and sometimes still are) called linear programming (LP) problems. We will use the word "programming" for common terms like LP, but favor the use of the term optimization over programming as it is more commonly used today and avoids confusion with modern computer code programming.

For functions that involve the solution of more complicated numerical models, it is usually not possible 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 would require infinite computational resources. However, to prove multimodality, one just needs to find two distinct local minima. Therefore, *a minimum should be presumed to be global until proven otherwise*. Depending on how sure we want to be about this hypothesis, we can perform multiple optimizations to try to find other local minima. As soon as a second minimum is found, the hypothesis is disproven, but any new optimization that converges to the same minimum adds evidence that supports this hypothesis and increases the probability that it is true.

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. 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 9. Given the definition above, we can infer that all multimodal functions are non-convex, but not all unimodal functions are convex (see Fig. 1.6).

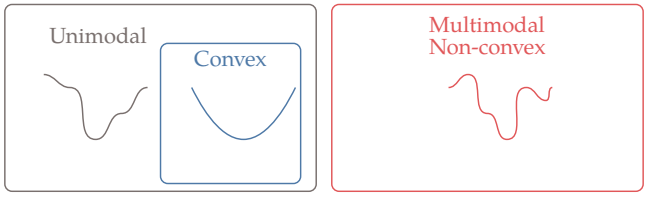


Figure 1.6: Multimodal functions have multiple minima, while unimodal functions have only one minimum. All multimodal functions are non-convex, but not all unimodal functions are convex.

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. 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 geometric properties of the beam may have uncertainty due to manufacturing deviations. For most of this text, we assume that functions are deterministic except in Chapter 11, where we provide an overview of this latter form of stochasticity under 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 at hand before deciding on which optimization algorithm to use. By "effective" algorithm, we mean first and foremost that the algorithm is capable of solving the problem at all, and secondly, that it does so reliably and efficiently. Figure 1.7 lists the attributes for the classification of optimization algorithms, which we discuss in more detail below. These attributes are often amalgamated, but they are actually 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 very specific problems where a certain 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 2.4, while MDO architectures are covered in Chapter 8.

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 6. 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 3 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 point that the optimizer converged to satisfies mathematical optimality conditions, something that is difficult to establish 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 that is provided by the user and the order of information that is actually used in the algorithm. For

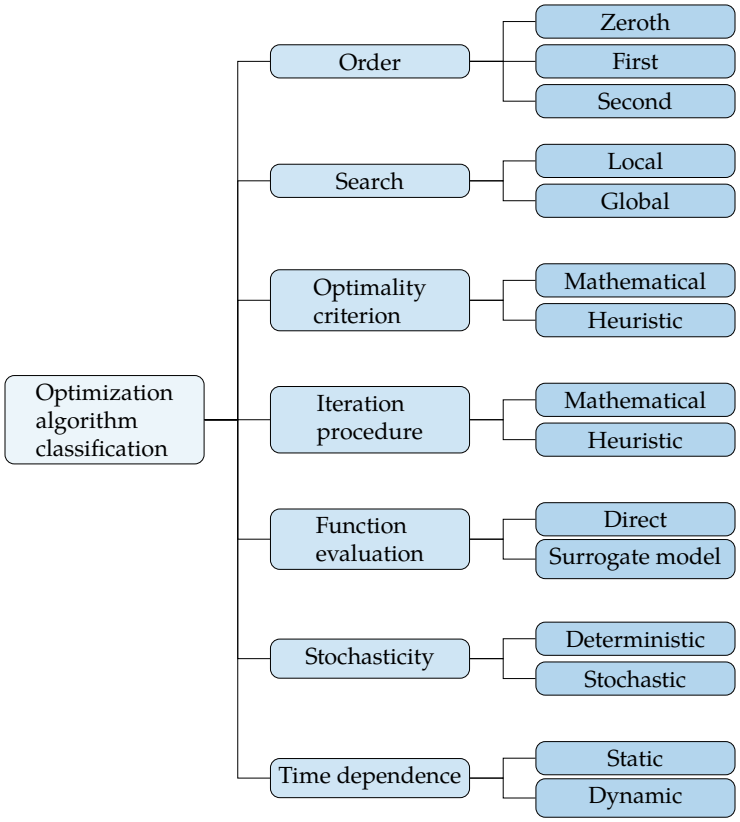


Figure 1.7: Optimization algorithms can be classified by using the attributes on the rightmost column. As with the problem classification, these attributes are independent and any combination is possible.

example, a user might only provide function values to a gradient-based algorithms 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). In practice, however, 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 a rigorous mathematical criterion for optimality. We also cover in detail the various approaches for computing gradients because the accurate and efficient computation of these gradients is crucial for the efficacy and efficiency of these methods (Chapter 4).

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 on their own 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 functions that are smooth enough, they require more knowledge about the models and optimization algorithm than gradient-free methods. Part of this book is devoted toward 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. Local methods, in spite of the name, are able to 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 very specific cases.

The classification of local versus global search 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 when it comes to how much of an algorithm's iterative process and optimality criteria is based on provable mathematical principles versus heuristics. The iterative process determines the sequence of points that get evaluated in the process of seeking the optimum, while the optimality criteria determines 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 is in contrast to 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 requires 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 can 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 largely 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 12.

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 in the machine

learning community, such as stochastic gradient descent (see brief overview in Further Notes at the end of the chapter).

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 would be 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 a sequence of optimization problems at different time instances because we must make decisions as time progresses. These are called *dynamic optimization problems* (also known as dynamic programming). In such problems, we solve a sequence of optimization problems where the decisions are design variables and the decisions at a time instance are 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 instance such that the overall time in a race course 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 covered in this book. They require an all together different approach, but still use many of the concepts covered here.

1.5 Timeline of Historical Developments

Optimization has a long history that started with geometry problems solved by the ancient Greek mathematicians. The invention of calculus much later open the door to many more problems, and the advent of numerical computing increased the range of problems that could be solved both in terms of type and scale. Some of the items below are linked to sections in the book when relevant.

300 bc: Euclid considers the minimal distance between a point a line and proves that a square has the greatest area among the rectangles with given total length of edges.

200 bc: Zenodorus works on "Dido's Problem", which involved finding the figure bounded by a line that has the maximum area for a given perimeter.

100 bc: Heron proves that light travels between two points through the path with shortest length when reflecting from a mirror, resulting in an angle of reflection equal to the angle of incidence.

1615: Kepler finds the optimal dimensions of wine barrel. He also formulated an early version of the "marriage problem" (an application of dynamic programming using the theory of optimal stopping) when he started to look for his second wife. The problem involved maximizing a utility function based on the balance of virtues and drawbacks of 11 candidates.

1621 Snellius discovers the law of refraction. This law follows the more general *principle of least time* (or Fermat's principle), which states that a ray of light going from one point to another will follow the path that takes the least time.

1646: Fermat shows that the gradient of a function is zero at an extreme point.

1695: Newton solves for the shape of a symmetrical body of revolution that minimizes fluid drag using calculus of variations.

1696: Johann Bernoulli challenges all the mathematicians in the world to find the path of a body subject to gravity that minimizes the travel time between two points of different heights—the brachistochrone problem. Bernoulli already had a solution that he kept secret. Five mathematicians respond with solutions: Newton, Jakob Bernoulli (Johann's brother), Leibniz, von Tschirnhaus, and l'Hôpital. Newton reportedly started solving the problem as soon as he received it, did not sleep that night, and took almost 12 hours to solve it, sending back the solution that same day.

1740: Euler starts the field of calculus of variations.

1746: Maupertuis proposes the principle of least action, which unifies various laws of physical motion. This is the precursor of the variational principle of stationary action, which uses calculus of variations and plays a central role in Lagrangian and Hamiltonian classical mechanics.

1784: Monge investigates a combinatorial optimization problem known as the transportation problem.

1805: Legendre describes the method of least squares, which was used in the prediction of asteroid orbits and curve fitting. Frederich Gauss publishes 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 engage in a bitter dispute on who first developed the method.

1815: Ricardo publishes the law of diminishing returns for land cultivation.

1847: Cauchy develops the steepest descent method, the first gradient-based method.

1857: Gibbs shows that chemical equilibrium is attained when the energy is a minimum.

1902: Farkas presents an important lemma that is later used in the proof of the Karush–Kuhn–Tucker theorem.

1917: Hancock publishes the first textbook on optimization: "Theory of Minima and Maxima".

1932: Menger presents a general formulation of the traveling salesman problem, one of the most intensively studied problems in optimization.

1939: Karush derives the necessary conditions for the inequality constrained problem in his Masters thesis. Kuhn and Tucker rediscover these conditions and publish their seminal paper in 1951. These became known as the Karush–Kuhn–Tucker (KKT) conditions. (Section 5.2)

1939: Kantorovich develops a technique to solve linear optimization problems after having been given the task of optimizing production in the Soviet government plywood industry.

1947: Dantzig publishes the simplex algorithm. Dantzig, who worked for the U.S. Air Force, reinvented and developed linear programming further to plan expenditures and returns in order to reduce costs to the army and increase losses to the enemy in World War II. The algorithm was kept secret until its publication.

1947: von Neumann develops the theory of duality for linear problems.

1949: The first international conference on optimization, the International Symposium on Mathematical Programming, is held in Chicago.

1951: Markowitz presents his portfolio theory that is based on quadratic optimization. He receives the Nobel memorial prize in economics in 1990.

1951: Box and Wilson introduce polynomial response surfaces.

1951: Krige proposes a stochastic process surrogate model now known as Kriging.

1954: Ford and Fulkerson research network problems, founding the field of combinatorial optimization.

1957: Bellman presents the necessary optimality conditions for dynamic programming problems. The Bellman equation was first applied to engineering control theory, and subsequently became an important principle in the development of economic theory.

1959: Davidon develops the first quasi-Newton method for solving nonlinear optimization problems. Fletcher and Powell publish further developments in 1963. (Section 3.5.4)

1960: Zoutendijk presents the methods of feasible directions to generalize the simplex method for nonlinear programs. Rosen, Wolfe, and Powell develop similar ideas.

1963: Wilson invents the sequential quadratic programming method for the first time. Han re-invents it in 1975 and Powell does the same in 1977.

1975: Pironneau [1] publishes a seminal paper on aerodynamic shape optimization, which first proposes the use of adjoint methods for sensitivity analysis. (Section 4.7)

1975: Holland proposes the first genetic algorithm. (Section 6.4)

1977: Haftka publishes one of the first MDO applications, which was aircraft wing design [2].

1979: Kachiyan proposes the first polynomial time algorithm for linear problems. The New York Times publishes the front headline "A Soviet Discovery Rocks World of Mathematics", saying, "A surprise discovery by an obscure Soviet mathematician has rocked the world of mathematics and computer analysis ... Apart from its profound theoretical interest, the new discovery may be applicable in weather prediction, complicated industrial processes, petroleum refining, the scheduling of workers at large factories ... the theory of secret codes could eventually be affected by the Russian discovery, and this fact has obvious importance to intelligence agencies everywhere."

1984: Karmarkar starts the age of interior point methods by proposing a more efficient algorithm for solving linear problems. In a particular application in communications network optimization, the solution time was reduced from weeks to days, enabling faster business and policy decisions. Karmarkar's algorithm stimulated the development of several other interior point methods, some of which are used in current codes for solving linear programs.

1985: The first conference in MDO, the Multidisciplinary Analysis and Optimization (MA&O) conference, takes place.

1988: Jameson [3] develops adjoint-based aerodynamic shape optimization for computational fluid dynamics (CFD).

1990 Sobieszczanski-Sobieski [4] proposes a formulation for computing the derivatives for coupled systems.

1994: Sandia National Laboratories initiates DAKOTA, an open source research toolkit for optimization and uncertainty quantification. (Chapter 11)

1995: Kennedy and Eberhart [5] invent the particle swarm optimization (PSO) algorithm.

1996: Braun and Kroo [6] develop the collaborative optimization MDO architecture, aiming to solve large-scale distributed optimization problems. (Section 8.3.2)

2004: Grant and Boyd [7] propose the disciplined convex programming methodology to construct convex problems and convert them to a solvable form. The following year they released CVX, a software program to solve these problems.

2005: Martins et al. [8] extend the adjoint method for computing the derivatives of coupled systems and demonstrate it on a wing design optimization problem [9].

2010: NASA Glenn releases OpenMDAO, an open source framework supporting multidisciplinary optimization [10].

2018: Martins and Hwang [11] present a unified theory for derivative computation techniques and an MDO architecture to enable efficient solution of coupled systems and computation of coupled derivatives [12]. (Section 4.9, Section 8.2.4)

1.6 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 to understand the characteristics of the optimization algorithms to select the appropriate algorithm to solve the problem.

1.7 Further Notes

- Nahin [13] provides a detailed history of optimization.
- Christian and Griffiths [14] explains a few practical applications of optimization to decisions in life.

- MDO emerged in the 1980s, following the success of the application of numerical optimization techniques to structural design [15, 16]. Aircraft design was one of the first applications of MDO because there is much to be gained by the simultaneous consideration of the various disciplines involved (such as structures, aerodynamics, propulsion, stability and controls), which are tightly coupled. The first applications focused on coupling the aerodynamics and structures in wing design [17, 2, 18, 19].

- Ashley [20] presents early applications of optimization to aircraft design and Sobieszczanski-Sobieski and Haftka [21] present MDO developments and applications up to the late 1990s. A more recent review focuses on MDO architectures, which we will describe in more detail in Chapter 8 [22]. There have also been various reviews on design optimization for various disciplines and applications, such as structural optimization [23, 24, 25], trajectory optimization [26], rotorcraft design optimization [27] and wind turbines [28].

- In the context of time dependence, we mentioned dynamic optimization and optimal control problems. There are numerous textbooks on this topic [29, 30].

- The main text mentioned stochastic gradient-based methods, which are widely used in the machine learning community. These algorithms are typically used on objectives of the form:

$$f(x) = \sum_{i=1}^{n} f_i(x) \tag{1.2}$$

  where f_i is an objective (e.g., an error to be minimized) for the i^{th} data point in the training set, and the total objective is summed over all training data. When the amount of training data is very large then computing the true gradient is too expensive. Instead, a random subset ($\mathcal{S}$) of training examples is used to estimate the gradient.

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \sum_{i} \frac{\mathrm{d}f_i}{\mathrm{d}x} \text{ for } i \in \mathcal{S} \tag{1.3}$$

  where $\mathcal{S}$ is some subset of the indices $[1 \ldots n]$. This approach works well for these specific problems because of the unique form for the objective.

Bibliography

[1] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64(01):97, 1974. ISSN 0022-1120. doi:10.1017/S0022112074002023.

[2] Raphael T. Haftka. Optimization of flexible wing structures subject to strength and induced drag constraints. *AIAA Journal*, 15(8):1101–1106, 1977. doi:10.2514/3.7400.

[3] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3):233–260, September 1988. doi:10.1007/BF01061285.

[4] Jaroslaw Sobieszczanski-Sobieski. Sensitivity of complex, internally coupled systems. *AIAA Journal*, 28(1):153–160, 1990. doi:10.2514/3.10366.

[5] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume IV, pages 1942–1948, Piscataway, NJ, 1995.

[6] R. D. Braun and I. M. Kroo. Development and application of the collaborative optimization architecture in a multidisciplinary design environment. In N. Alexandrov and M. Y. Hussaini, editors, *Multidisciplinary Design Optimization: State of the Art*, pages 98–116. SIAM, 1997. URL `citeseer.nj.nec.com/braun96development.html`.

[7] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[8] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62, March 2005. doi:10.1023/B:OPTE.0000048536.47956.62.

[9] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, May 2004. doi:10.2514/1.11478.

[10] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019. doi:10.1007/s00158-019-02211-z.

[11] Joaquim R. R. A. Martins and John T. Hwang. Review and unification of methods for computing derivatives of multidisciplinary computational models. *AIAA Journal*, 51(11):2582–2599, November 2013. doi:10.2514/1.J052184.

[12] John T. Hwang and Joaquim R. R. A. Martins. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM Transactions on Mathematical Software*, 44(4):Article 37, June 2018. doi:10.1145/3182393.

[13] Paul J. Nahin. *When least is best: how mathematicians discovered many clever ways to make things as small (or as large) as possible*. Princeton University Press, 2004. ISBN 0-691-07078-4.

[14] Brian Christian and Tom Griffiths. *Algorithms to live by: The computer science of human decisions*. Macmillan, 2016.

[15] L A Schmit. Structural Design by Systematic Synthesis. In *2nd Conference on Electronic Computation*, pages 105–132, New York, NY, 1960. ASCE.

[16] A. Schmit Jr., L. Structural synthesis—its genesis and development. *AIAA Journal*, 19(10):1249–1263, October 1981. doi:10.2514/3.7859.

[17] Raphael T. Haftka. Automated procedure for design of wing structures to satisfy strength and flutter requirements. Technical Report TN D-7264, NASA, July 1973.

[18] R.T. Haftka. Structural optimization with aeroelastic constraints—-a survey of U.S. applications. *International Journal of Vehicle Design*, 7:381–392, 1986.

[19] B. Grossman, Z. Gurdal, G. J. Strauch, W. M. Eppard, and R. T. Haftka. Integrated aerodynamic/structural design of a sailplane wing. *Journal of Aircraft*, 25(9):855–860, 1988. doi:10.2514/3.45670.

[20] Holt Ashley. On making things the best — aeronautical uses of optimization. *Journal of Aircraft*, 19(1):5–28, January 1982.

[21] J. Sobieszczanski-Sobieski and R. T. Haftka. Multidisciplinary aerospace design optimization: Survey of recent developments. *Structural Optimization*, 14(1):1–23, 1997. doi:10.1007/BF011.

[22] Joaquim R. R. A. Martins and Andrew B. Lambe. Multidisciplinary design optimization: A survey of architectures. *AIAA Journal*, 51(9):2049–2075, September 2013. doi:10.2514/1.J051895.

[23] Raphael T. Haftka and Ramana V. Grandhi. Structural shape optimization—a survey. *Computer Methods in Applied Mechanics and Engineering*, 57(1):91–106, 1986. ISSN 0045-7825. doi:10.1016/0045-7825(86)90072-1.

[24] Kazuhiro Saitou, Kazuhiro Izui, Shinji Nishiwaki, and Panos Papalambros. A survey of structural optimization in mechanical product development.

Journal of Computing and Information Science in Engineering, 5(3):214–226, 09 2005. ISSN 1530-9827. doi:10.1115/1.2013290.

[25] G. I. N. Rozvany. A critical review of established methods of structural topology optimization. *Structural and Multidisciplinary Optimization*, 37(3): 217–237, Jan 2009. ISSN 1615-1488. doi:10.1007/s00158-007-0217-0.

[26] John T Betts. Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, 21(2):193–207, 1998.

[27] Ranjan Ganguli. Survey of recent developments in rotorcraft design optimization. *Journal of Aircraft*, 41(3):493–510, 2004. doi:10.2514/1.58.

[28] Adam Chehouri, Rafic Younes, Adrian Ilinca, and Jean Perron. Review of performance optimization techniques applied to wind turbines. *Applied Energy*, 142:361–388, mar 2015. doi:10.1016/j.apenergy.2014.12.043.

[29] A. E. Bryson and Y. C. Ho. *Applied Optimal Control; Optimization, Estimation, and Control*. Blaisdell Publishing, 1969.

[30] Dimitri P. Bertsekas. *Dynamic programming and optimal control*. Athena Scientific, Belmont, MA, 1995.

Exercises

1. *Using an unconstrained optimizer*. The Rosenbrock function is a good starter problem for optimization. It is only has two-dimensions so can be easily visualized, but still provides some challenge as the optimum lays in a narrow curved valley with steep gradients to either side. The objective function is:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

   Use an existing software package to solve this problem. Did the optimizer converge? What is the optimal solution? Try using different starting points. What happens if you use a starting point really far away from the optimum: $[x_0, y_0] = [100, 100]$?

2. *Using a constrained optimizer*. Building on the previous problem let us now add a couple of constraints. The objective is the same, but we will add two inequality constraints:

$$x^2 + y^2 \leq 1$$
$$x + 3y \leq 5$$

   and bound constraints:

$$-5 \leq x \leq 5$$
$$-5 \leq y \leq 5$$

Use an existing software package to solve this problem. Why does the solution differ from the previous problem? Which of the inequality constraints were active at the solution?

CHAPTER 2

Numerical Models and Solvers

In the previous chapter, we discussed function characteristics from the point of view of the function's output—the black box view shown in Fig. 1.4. 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. It is also important to understand the errors involved in the modeling process so that we can interpret optimization results correctly. We also introduce *multidisciplinary* models and the methods available to solve them.

2.1 Modeling Process and Errors

The steps in the modeling process are shown in Fig. 2.1. The physical system represents the reality that we want to *model*. The mathematical model—also known as *governing equation*—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 *coded* using a computer programming language to obtain 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, so the total error is the sum of modeling errors, discretization errors, coding errors, and numerical errors. Most textbooks are too optimistic and do not explicitly list coding errors as a possibility. Validation and verification processes enable us to quantify these errors. Verification is concerned with making sure the errors

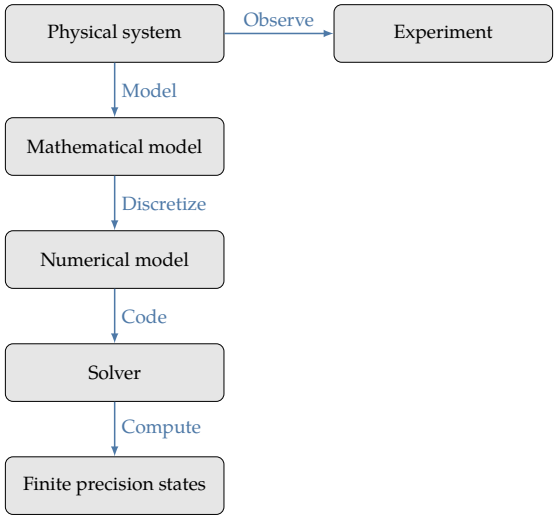


Figure 2.1: Physical problems are modeled and then solved numerically to produce function values.

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 consists in 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).

2.2 Types of Models

Mathematical models vary greatly in complexity and scale. In the simplest case, a model can be written as one or more explicit algebraic equations. The algebraic equations in a model could also be implicit, which complicates their solution because it requires an iterative algorithm to solve them.

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 in the domain, then we have a dynamic model. When a differential equations is defined in a domain defined by only one variable (1D spatial 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. Mathematical models can also include integrals, which can be discretized with quadrature rules.

The final result of the discretization process is a set of algebraic equations. This is a potentially large set of equations depending on the domain and discretization (it is common to have millions of equations in steady 3D computational fluid dynamic problems). The number of unknowns (the *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.

No mater how complex the mathematical model and how involved the discretization, a numerical model can always be written as a set of algebraic equations with a zero right-hand side,

$$R(x, u) = 0, \tag{2.1}$$

which a vector of n_u *residuals* corresponding to the discretized governing equations. Here, u is the vector of state variables that are determined by solving these equations for a given design (x). This set of equations typically needs to be solved iteratively and defines u as an implicit function of x.

This residuals notation can still be used to represent explicit functions, so we can use it without loss of generality. For a given explicit function $h(x)$, we can write $R(x, u_h) = h(x) - u_h = 0$, where u_h is a state variable that we introduced that is associated with h. Using this formulation, we compute the explicit function $h(x)$ and either let the solver for the complete system solve for u_h together with the other state variables, or simply set $u_h = h(x)$. While this notation might seem contrived, it will be useful for us in later chapters.

Example 2.1. Implicit equations.

Implicit equations are common in engineering applications. Consider, for example, the equation:

$$\frac{\lambda}{m} + \cos\lambda = 0 \tag{2.2}$$

This is an *implicit* algebraic equation where the state variable is λ. Sometimes we can rearrange an implicit function to make it *explicit*, but in this case we cannot solve this equation explicitly for λ.

2.3 Types of Solvers

There are a number of methods available to solve the discretized governing equations (2.1). Here we will ignore x, because we want to solve the governing equations for a fixed set of design of design variables. Our objective is to find the state variables u such that $R(u) = 0$. There are two main types of methods, depending on whether the equations to be solved are linear or nonlinear, as shown in Fig. 2.2.

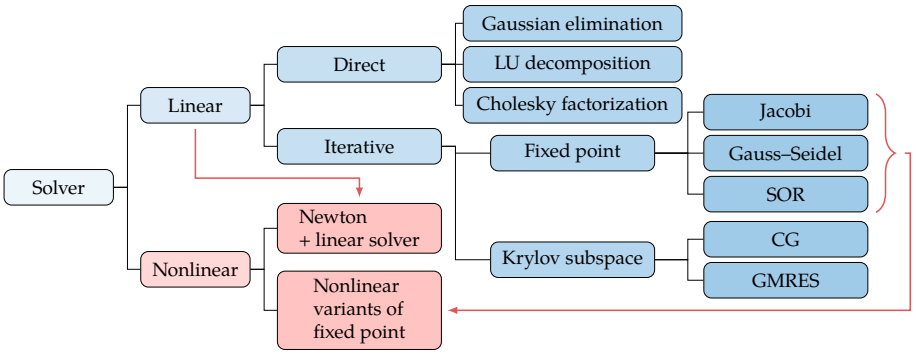


Figure 2.2: Overview of solution methods for linear and nonlinear systems.

If the equations are linear, they can be written as

$$R(u) = b - Au = 0, \tag{2.3}$$

where A is a square $(n_u \times n_u)$ 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.

The direct methods include Gaussian elimination and factorizations like LU decomposition. Cholesky factorization is a direct method specialized for the case where the matrix A is symmetric and positive definite.

While direct methods provide the precise solution at the end of a procedure that runs only once, iterative methods repeat a procedure that provides an approximate solution that is improved for every iteration. Iterative methods are advantageous for large systems of equations because they require less memory. These methods are divided into fixed-point iteration methods (also known as stationary iterative methods) and Krylov subspace methods.

Fixed-point methods generate a sequence of iterates $u_1, \ldots, u_k, \ldots$ using a function

$$u_{k+1} = G(u_k), \quad k = 0, 1, \ldots, \tag{2.4}$$

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

$$u = M^{-1}(Nu + b). \tag{2.5}$$

Since $Nu = Mu - Au$, substituting this in to the above equation results in the iteration

$$u_{k+1} = u_k + M^{-1}(b - Au_k) = u_k + M^{-1}R(u_k). \tag{2.6}$$

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,

$$u_{k+1} = u_k + D^{-1}R(u_k). \tag{2.7}$$

In component form, this can be written as

$$u_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1, j\neq i} a_{ij}u_j^{(k)}\right], \quad i = 1, \ldots, n_u \tag{2.8}$$

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

$$u_{k+1} = u_k + E^{-1}R(u_k), \tag{2.9}$$

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

$$u_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j<i} a_{ij}u_j^{(k+1)} - \sum_{j>i} a_{ij}u_j^{(k)}\right], \quad i = 1, \ldots, n_u. \tag{2.10}$$

The successive over-relaxation (SOR) method uses an update that is a weighted average of the Gauss–Seidel update and the previous iteration,

$$u_{k+1} = u_k + \omega\left[(1-\omega)D + \omega E\right]^{-1}R(u_k), \tag{2.11}$$

where ω is a scalar between one and two. Setting $\omega = 1$ above yields the Gauss–Seidel method. SOR in component form is

$$u_i^{(k+1)} = (1-\omega)u_i^{(k)} + \frac{\omega}{a_{ii}}\left[b_i - \sum_{j<i} a_{ij}u_j^{(k+1)} - \sum_{j>i} a_{ij}u_j^{(k)}\right], \quad i = 1, \ldots, n_u. \tag{2.12}$$

Newton's method is the basis for many nonlinear equation solvers and can be derived as follows. Using the Taylor series expansion generalized for vector variables (u) and vector-valued functions (R) about a state u_k, we can write the residuals for a state $u_{k+1} = u_k + \Delta u_k$ as

$$R(u_{k+1}) = R(u_k) + \left.\frac{\partial R}{\partial u}\right|_{u=u_k} \Delta u_k + O(\Delta u_k^2) + \ldots. \tag{2.13}$$

Since we want to find u such that $R(u) = 0$, we set the above equation to zero. Then, ignoring the higher order terms, we can write

$$\left.\frac{\partial R}{\partial u}\right|_{u=u_k} \Delta u_k = -R(u_k), \quad k = 0, 1, \ldots, \tag{2.14}$$

where $\partial R/\partial u$ is a square matrix ($n_u \times n_u$) that represents a linearization of the nonlinear equations. This defines an iterative method where the matrix of partial derivatives is updated at every iteration. The right-hand side is also updated at every iteration and is the residual of the nonlinear equations evaluated for the last available state. The Newton system (2.14) is linear and can be solved using any of the methods for solving linear systems mentioned above, as indicated by the arrow in Fig. 2.2.

The fixed-point iteration methods can also be used to solve nonlinear systems. Fixed-point iteration methods are not efficient, converging linearly at best. Newton methods tend to be much more efficient and near the solution they exhibit quadratic convergence, which is a desirable characteristic. On the other hand, Newton's method requires a good initial guess and does not always converge to the solution. There are a number of variants of Newton's method that make it more robust.

Some of these methods can take advantage of systems that are *sparse*, that is when each equation involves only a subset of all the state variables and therefore many entries in A are zeros.

From a mathematical point of view, the model governing equations (2.1) 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 method*, and is also known as *simultaneous analysis and design* (SAND). In this book, however, we assume that the governing equations are solved separately for each optimization iteration.

Ultimately, we need to compute the values of the objective and constraint functions. While it is possible that these happen to be among the state variables, they are usually *explicit* functions of the state and design variables. The dependency of these functions on the state and design variables is illustrated in Fig. 2.3.

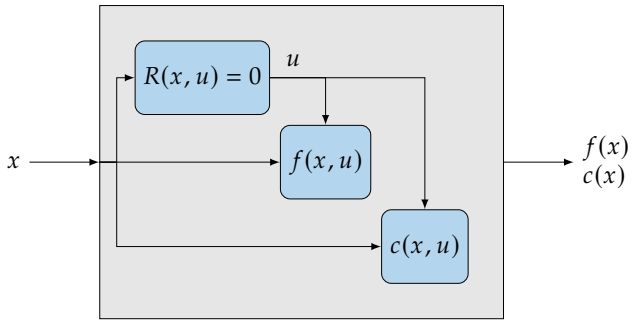


Figure 2.3: The computation of the objective (f) and constraint (c) functions for a given set of design variables (x) often involves the solution of the numerical model ($R = 0$) with respect to state variables (u).

Example 2.2. A common situation requiring the solution of implicit equations.

One common structural optimization problem is weight minimization where the stresses in the structure are constrained to be under a certain value (usually with some margin) to avoid failure.
In a linear finite-element structural analysis, the displacements are computed by solving,

$$Ku = F, \tag{2.15}$$

where K is the stiffness matrix, F is the vector of the external forces, and u are the structural displacements, which are the states in this case. This linear system is an implicit set of equations that can be written as a set of governing equation residuals

$$R(x,u) = Ku - F = 0, \tag{2.16}$$

where the design variables x are fixed for a given analysis. The design variables can be the shape of the structure, the sizing (thickness or cross-sectional area) of the elements, or both. The structural stresses are explicitly computed from the displacements and given by a linear relationship

$$\sigma = Su. \tag{2.17}$$

The structural weight is an explicit function of the sizing and shape variables, and does not involve the solution of the governing equations at all.
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, as shown in Fig. 2.4.

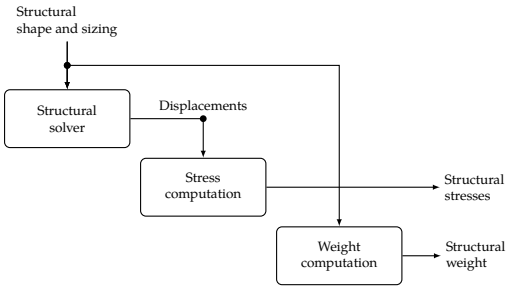


Figure 2.4: Numerical model for a FEM-based structural sizing optimization problem.

2.4 Multidisciplinary Models

As mentioned in Chapter 1, many engineering systems are multidisciplinary, which means that the system is composed of different disciplines that are coupled. Throughout this book, instead of "discipline", we will also use the term *component* because it is more general. When components in a system represent different physics, the term "multiphysics" is commonly used.

In general, these models would be coupled as shown in Fig. 2.5 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.

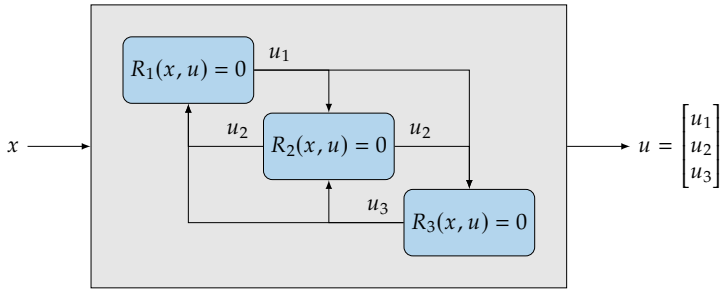


Figure 2.5: Multidisciplinary model composed of three numerical models. Each model solves for its own state variable vector (u_1, u_2, u_3) but in general requires the other state vectors as inputs. This set of models would replace the single model in Fig. 2.3.

Example 2.3. Defining a multidisciplinary optimization problem.

Let us expand on Example 2.2 to consider a multidisciplinary numerical model for an aircraft wing, where the aerodynamics and structures disciplines are coupled. 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. 2.6. 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 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.

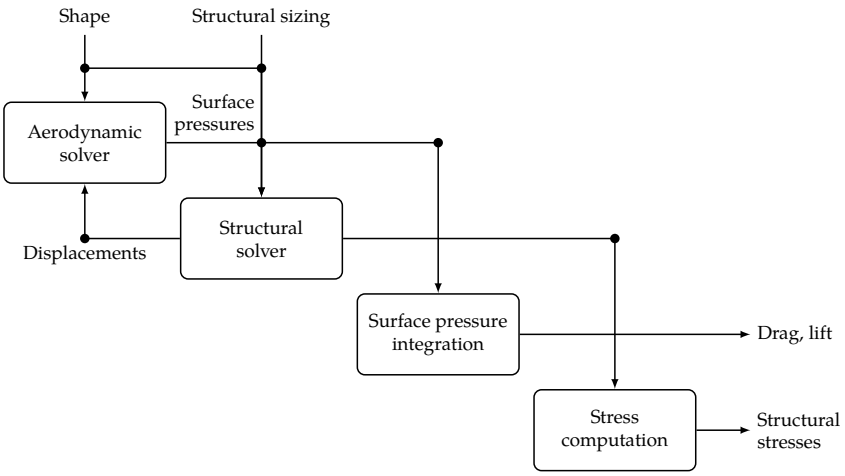


Figure 2.6: Multidisciplinary numerical model for an aircraft wing.

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 Sec. 8.2.4).

2.4.1 Components

We prefer to use the more general term *components* 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 initial or boundary 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. As we will see in Chapter 8, 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

$$R(u) = 0 \Leftrightarrow \begin{cases} R_1(u_1, \ldots, u_i, \ldots, u_n) = 0 \\ \vdots \\ R_i(u_i, \ldots, u_i, \ldots, u_n) = 0 \\ \vdots \\ R_n(u_1, \ldots, u_i, \ldots, u_n) = 0 \end{cases}, \tag{2.18}$$

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. 2.5.

Components can be either implicit or explicit, a concept we introduced in Section 2.2. To solve an implicit component, we need an algorithm for driving the equation residuals, $R_i(u_1, \ldots, u_i, \ldots, u_n) = 0$, 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,

$$R_i(u_1, \ldots, u_n) = u_i - g(u_j) = 0 \quad \forall j \neq i. \tag{2.19}$$

Most disciplines involve a mix of implicit and explicit components because, as mentioned in Section 2.2 and shown in Fig. 2.3, 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.

2.4.2 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. 2.7. 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.

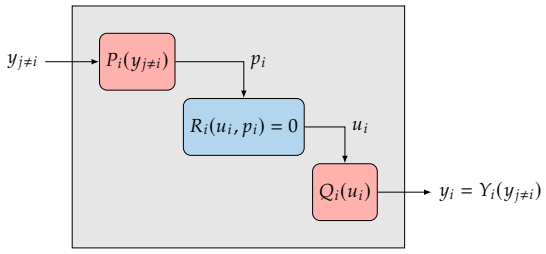


Figure 2.7: In the general case, a solver includes a conversion of inputs and outputs distinct from its states.

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. 2.7 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

$$y = Y(y) \Leftrightarrow \begin{cases} y_1 = Y_1(y_2, \ldots, y_n) \\ \quad \vdots \\ y_i = Y_i(y_{j\neq i}) \\ \quad \vdots \\ y_n = Y_i(y_1, \ldots, y_{n-1}) \end{cases}, \tag{2.20}$$

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

$$R_i = y_i - Y_i(y_{j \neq i}) = 0, \tag{2.21}$$

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. (2.20). 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. (2.19). The result is a component with three subcomponents as shown in Fig. 2.8. 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.

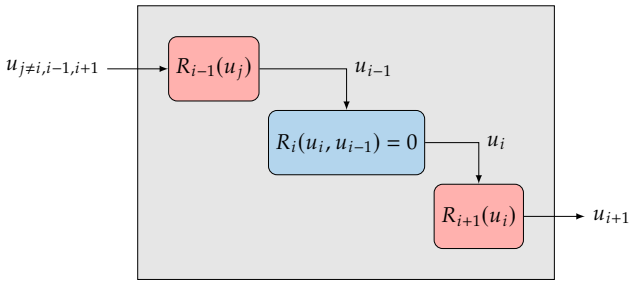


Figure 2.8: The translation of inputs and outputs can be represented as components with its own state variables, so any coupled system can be written as $R(u) = 0$.

An example of a system with three solvers is shown in Fig. 2.9. 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 some solvers have exposed residuals and states, while others do not. Furthermore, there might be translation functions between black boxes that are exposed.

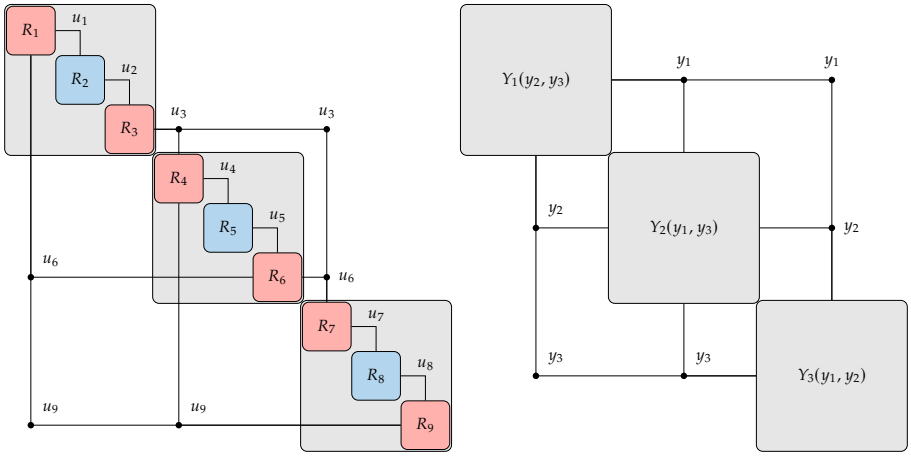


Figure 2.9: Two system-level views of coupled system with three solvers: all components exposed and written as residuals and states (left) and black box representation where only inputs and outputs for each solver are visible (right), where $y_1 \equiv u_3$, $y_2 \equiv u_6$, and $y_3 \equiv u_9$.

2.4.3 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 on the left in Fig. 2.10. 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.[1] As shown in Fig. 2.10, there are in general off-diagonal entries both above and below the diagonal, where the entries above feed forward, while entries below feed backward.

The mathematical representation of these dependencies is given by a graph (center of Fig. 2.10), 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. 2.10, 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* (right in Fig. 2.10), which has the same

[1] In some of the DSM literature this definition is reversed, where "row" and "column" are interchanged, resulting in a transposed matrix.

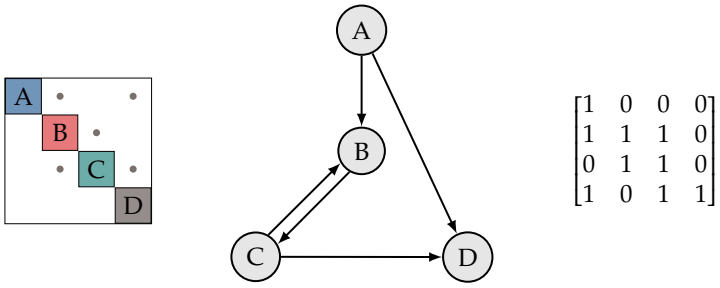


Figure 2.10: Design structure matrix (left), directed graph (center) and adjacency graph (left) showing the dependencies of an hypothetical system.

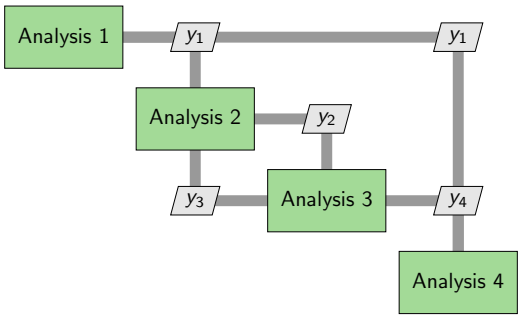


Figure 2.11: XDSM showing data dependencies for the four-component coupled system of Fig. 2.10.

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. 2.11 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.

2.4.4 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 fixed-point iteration in numerical linear algebra, but applied to the coupling variables at the system level.

The two basic fixed-point iteration methods are the Jacobi and Gauss–Seidel methods. The Jacobi methods requires a guess for all coupling variables to start with and calls for the solution of all components give 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 Algorithm 1. When applied to a system of components, we call it the block-Jacobi method, where "block" refers to each component.

The block-Jacobi method is also illustrated using an XDSM in Fig. 2.12 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 $2 \rightarrow 1$).

Algorithm 1 Nonlinear Block-Jacobi algorithm.

Input: $u^{(0)} = [u_1^{(0)}, \ldots, u_n^{(0)}]$ ▷ Guesses for coupling variables
  $k = 1$
  while $||u^{(k)} - u^{(k-1)}||_2 < \epsilon$ **do**
    for all $i \in \{1, \ldots, n\}$ **do** ▷ Can be done in parallel
      Solve $R_i\left(u_i^{(k)}, u_j^{(k-1)}\right) = 0$ to get $u_i^{(k)}$, where $j \neq i$
    end for
    $k = k + 1$
  end while
Return: $u = [u_1, \ldots, u_n]$ ▷ System-level states

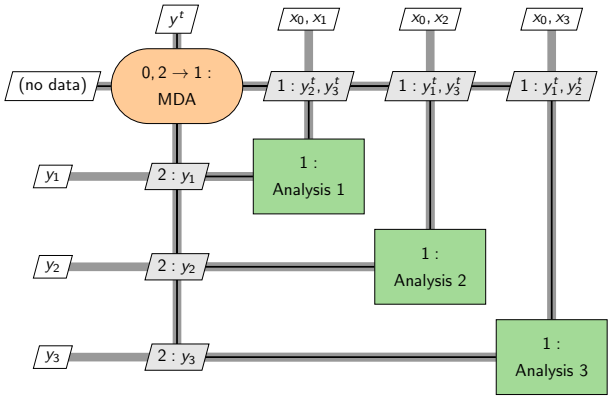


Figure 2.12: A block Jacobi multidisciplinary analysis process to solve a three-component coupled system.

The nonlinear block Gauss–Seidel algorithm is similar to the Jacobi one. 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 $i = 1, \ldots, n$ 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. 2.13 and formalized in Algorithm 2.

Algorithm 2 Nonlinear block Gauss–Seidel algorithm.

Input: $u^{(0)} = [u_1^{(0)}, \ldots, u_n^{(0)}]$ ▷ Guesses for coupling variables
  $k = 1$
  while $||u^{(k)} - u^{(k-1)}||_2 < \epsilon$ **do**
    for $i = 1, n$ **do**
      Solve for $u_i^{(k)}$ such that $R_i\left(u_1^{(k)}, \ldots, u_{i-1}^{(k)}, u_i^{(k)}, u_{i+1}^{(k-1)}, \ldots, u_n^{(k-1)}\right) = 0$
    end for
    $k = k + 1$
  end while
Return: $u = [u_1, \ldots, u_n]$ ▷ System-level states

Both the block Jacobi and Gauss–Seidel methods converge linearly, but Gauss–Seidel converges more quickly because each equation uses the latest information available.

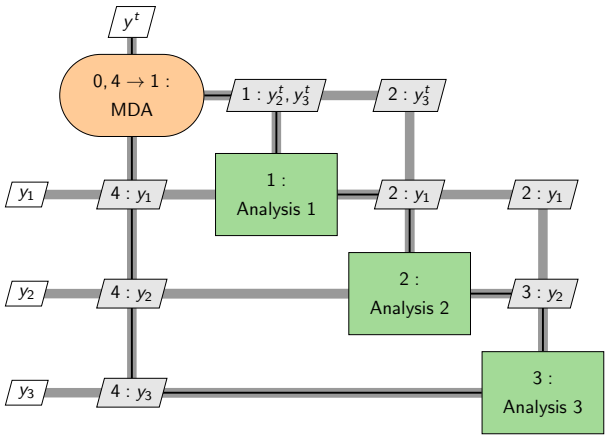


Figure 2.13: A block Gauss–Seidel multidisciplinary analysis (MDA) process to solve a three-discipline coupled system.

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

$$\left.\frac{\partial R}{\partial u}\right|_{u=u^{(k)}} \Delta u = -R\left(u^{(k)}\right), \tag{2.22}$$

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,

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial u_1} & \cdots & \frac{\partial R_n}{\partial u_n} \end{bmatrix} \begin{bmatrix} \Delta u_1 \\ \vdots \\ \Delta u_n \end{bmatrix} = - \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix}, \tag{2.23}$$

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 4. 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 Algorithm 3.

Algorithm 3 Newton method for system-level convergence.

Input: $u^{(0)} = [u_1^{(0)}, \ldots, u_n^{(0)}]$ ▷ Guesses for coupling variables
  $k = 1$
  while $||u^{(k)} - u^{(k-1)}||_2 < \epsilon$ **do**
    for all $i \in \{1, \ldots, n\}$ **do** ▷ Can be done in parallel
      Solve $R_i\left(u_i^{(k)}, u_j^{(k-1)}\right) = 0$ to get $u_i^{(k)}$, where $j \neq i$
      Compute $\dfrac{\partial R_i}{\partial u_j}$ for $j = 1, \ldots, n$
    end for
    Solve block Newton system (2.23) for Δu
    $u^{(k+1)} = u^{(k)} + \Delta u$
    $k = k + 1$
  end while
Return: $u = [u_1, \ldots, u_n]$ ▷ System-level states

Like the plain Newton method, this block 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.

2.5 Further Notes

- Ascher and Greif [1] is a more detailed introduction to the numerical methods mentioned in this chapter.
- Saad [2] is an authoritative reference for iterative methods that is especially useful for large-scale numerical models.
- Hwang and Martins [3] provide a more detailed description of methods to solve coupled numerical models.

Bibliography

[1] Uri M. Ascher and Chen Greif. *A first course in numerical methods*. SIAM, 2011.

[2] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2^{nd} edition, 2003.

[3] John T. Hwang and Joaquim R. R. A. Martins. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM Transactions on Mathematical Software*, 44(4):Article 37, June 2018. doi:10.1145/3182393.

CHAPTER 3

Unconstrained Gradient-Based Optimization

Among the problems discussed in the introduction, in this chapter we focus our attention on unconstrained minimization problems with continuous design variables. Most engineering design problems are constrained, but constrained optimization algorithms build on the techniques used for unconstrained optimization. We assume the objective function to be nonlinear and C^2 continuous. There is no assumption about multimodality and there are no guarantees that the algorithm finds the global optimum. Finally, we assume that the function is deterministic.

The optimization problem can be written as

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{with respect to} && x_i \qquad i = 1, \ldots, n_x. \end{aligned} \tag{3.1}$$

Recall that x contains the design variables that the optimization algorithm can change and that if we want to *maximize* f, we just need to replace the objective above with $-f(x)$.

Referring back to the optimization algorithm classification attributes (Fig. 1.7), 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.

3.1 Gradients, Hessians, and Taylor Series

Gradient-based optimization algorithms start from one guess and follow a path with discrete steps towards the optimum, as shown in Fig. 3.1. To determine

these steps, gradient-based methods need at the very least the gradient (first order information). Some methods also use the curvature (second order information). In addition, we will require *directional derivatives* when searching along a particular direction in n-dimensional space.

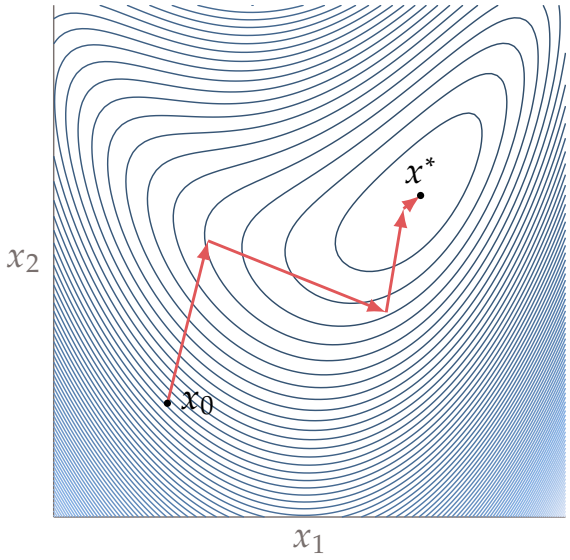


Figure 3.1: Gradient-based optimization follows a sequence of steps in n-dimensional space that converge to the optimum.

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

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]^T . \tag{3.2}$$

Each gradient vector component quantifies the local rate of change in the function with respect to the corresponding design variable. Physically, the gradient is a vector pointing in the direction of greatest function increase from the current point. Gradient vectors are perpendicular to the function contour lines, as shown in Fig. 3.2 for a 2D function. More generally, the gradient vectors are normal to contour surfaces of constant f in n-dimensional space.

The gradient components give the rate of change of the function in each of the coordinate directions (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 we can find this by projecting the gradient onto the

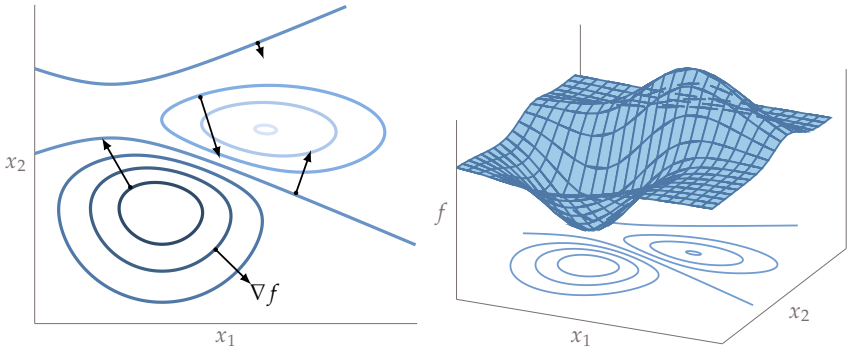


Figure 3.2: Gradient vectors indicate the direction of steepest increase and their magnitude represents the rate of that increase (left). The lighter contours are larger in value.

desired direction p using the dot product

$$D_p f(x) = \nabla f^T p. \tag{3.3}$$

An 2D example of this projection is shown in Fig. 3.3.

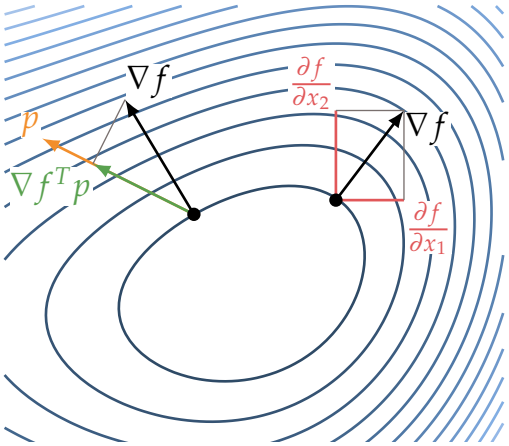


Figure 3.3: Projection of the gradient in an arbitrary direction p and components of the gradient vector in the 2D case.

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,

$$D_p f(x) = \nabla f^T p = \|\nabla f\| \, \|p\| \cos\theta, \tag{3.4}$$

we can see that this is maximized when $\theta = 0°$. That is, the directional derivative is largest when p points in the same direction as ∇f. If $-90° < \theta < 90°$, the directional derivative is positive and is thus in a direction of increase (Fig. 3.4). If $90° < \theta < 270°$, the directional derivative is negative and p points in a descent direction. Finally, if $\theta = \pm 90°$, the directional derivative is 0 and thus the function value does not change and is locally flat in that direction. That condition occurs if ∇f and p are orthogonal, and thus the gradient is always orthogonal to contour surfaces.

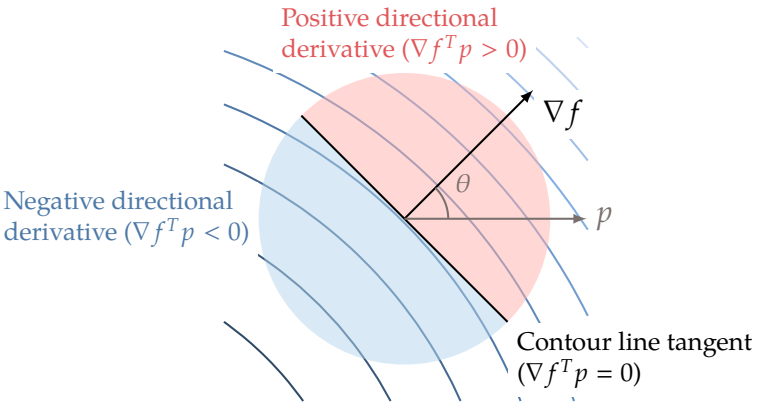


Figure 3.4: The gradient ∇f is always orthogonal to contour lines (surfaces), and the directional derivative in the direction of p is given by $\nabla f^T p$.

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 take the partial derivative of each gradient component j with respect to each coordinate direction i, that is

$$\frac{\partial^2 f}{\partial x_i \partial x_j}. \tag{3.5}$$

This yields a square $n \times n$ matrix of second-order partial derivatives called the *Hessian*:

$$\nabla^2 f(x) = H(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \tag{3.6}$$

If the function has continuous second partial derivatives, the order of differen-

tiation does not matter and the the mixed partial derivatives are equal,

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}. \tag{3.7}$$

Thus, the Hessian is a symmetric matrix with $n(n+1)/2$ independent elements.

The Hessian can be projected into an arbitrary direction p to find the rate of change in the gradient in that direction by taking the matrix-vector product Hp. This projected curvature is an n-vector that represents the direction and magnitude of the maximum rate of change in the projected gradient. To find the scalar corresponding to the curvature of the function along p (the second-order directional derivative), we need to project this vector onto p,

$$D_p^2 f(x) = p^T H p. \tag{3.8}$$

For any given Hessian in n dimensions, it is possible to find n directions v along which projected curvature aligns with that direction, that is,

$$Hv = \kappa v. \tag{3.9}$$

This is an eigenvalue problem whose eigenvectors represent the *principal curvature* directions and the eigenvalues κ quantify the corresponding curvatures.

As previously mentioned, the Taylor series expansion provides the foundation for gradient-based optimization algorithms. For a one-dimensional function we could use a Taylor series to obtain the approximation of a function using information at x to approximate the function at points $x + \delta x$:

$$f(x+\delta) = f(x) + f'(x)\delta + \frac{1}{2} f''(x)\delta^2 + O(\delta^3), \tag{3.10}$$

where we only use up to second-order terms. This corresponds to a quadratic approximation of the function, which we use extensively in gradient-based optimization.

For an n-dimensional function, the Taylor series can be applied in any direction. This can be done by projecting the gradient and Hessian onto the desired direction p, to get an approximation of the function at any nearby point $x + \alpha p$:

$$f(x+\alpha p) = f(x) + \alpha \nabla f(x)^T p + \frac{1}{2}\alpha^2 p^T H(x) p + O\left(\alpha^3\right). \tag{3.11}$$

3.2 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. According to the extreme value theorem, 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 a local neighborhood of x^*. As discussed above, a second-order Taylor-series expansion about x^* for small steps of size p is

$$f(x^* + p) = f(x^*) + \nabla f(x^*)^T p + \frac{1}{2} p^T H(x^*) p + \ldots. \tag{3.12}$$

For x^* to be an optimal point, we must have $f(x^* + p) \geq f(x^*)$ for all p or $f(x^* + p) - f(x^*) \geq 0$. This implies

$$\nabla f(x^*)^T p + \frac{1}{2} p^T H(x^*) p \geq 0. \tag{3.13}$$

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

$$\nabla f(x^*)^T p \geq 0. \tag{3.14}$$

Because p can be in any arbitrary direction, the only way this inequality can be satisfied is if all the elements of $\nabla f(x^*)$ are zero. Otherwise, we could choose a direction (such as $p = -\nabla f$) for which the inequality would not be satisfied, and we would therefore not be at an optimal point. This is the *first-order optimality condition*. In the one-dimensional case, this reduces to $f'(x^*) = 0$.

Since the gradient term is zero, we must now satisfy the remaining term,

$$p^T H(x^*) p \geq 0. \tag{3.15}$$

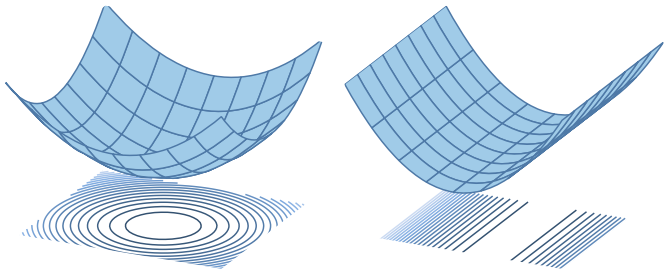
You may recognize this as the definition of a *positive semidefinite* matrix. In other words, the Hessian $H(x^*)$ must be positive semidefinite. Recall that the eigenvalues of a positive semidefinite matrix are all greater than or equal to zero. The one-dimensional analogue to this condition is $f''(x^*) \geq 0$.

These conditions on the gradient and curvature are *necessary conditions* for a local minimum. These conditions are not sufficient because if the curvature is zero in at least 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.

Sufficient conditions for optimality require that the curvature is positive in any direction. This is also referred to as a strong minimum. Mathematically, sufficient conditions require that H be *positive definite* ($p^T H(x^*) p > 0$). Figure 3.5 shows some examples of quadratic functions that are positive definite (all positive eigenvalues), positive semidefinite (nonnegative eigenvalues), indefinite (mixed eigenvalues), and negative definite (all negative eigenvalues).

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

$$\begin{aligned} &\nabla f(x^*) = 0, \\ &H(x^*) \text{ is positive semidefinite.} \end{aligned} \tag{3.16}$$



(a) Positive definite (b) Positive semidefinite

(c) Indefinite (d) Negative definite

Figure 3.5: Quadratic functions with different types of Hessians from positive definite to negative definite.

The *sufficient* optimality conditions are

$$\begin{aligned} &\nabla f(x^*) = 0, \\ &H(x^*) \text{ is positive definite.} \end{aligned} \tag{3.17}$$

Computationally, the entries in ∇f will not reach exactly zero because of finite-precision arithmetic. Instead we define convergence for the first criterion when

$$\left\|\nabla f\right\|_\infty < \tau, \tag{3.18}$$

where τ is some tolerance. A typical absolute tolerance is $\tau = 10^{-6}$ or a six-order magnitude reduction in gradient if we are using a relative tolerance. We typically use an infinity norm. The second condition, 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 3.5, our search directions are always descent directions and so if we reach a stationary point we know that it is a local minimum.

3.3 Two Overall Approaches to Finding the Optimum

In addition to determining whether a point is an optimum, the optimality conditions derived above also serve to find optimum points. Gradient-based optimization algorithms start with a guess, x_0, and generate a series of points, $x_1, x_2, \ldots, x_k, \ldots$, that converge to a local optimum, x^*, as previously illustrated in Fig. 3.1. At each iteration, some form of the Taylor series about the current point is used to find the next point. However, a truncated Taylor series is in general only a good model within a small neighborhood, and a *globalization* strategy is needed to ensure convergence to an optimum. Globalization here means to make the algorithm robust enough so that it is able to converge to a local minimum starting from any point in the domain (not to be confused with trying to find the global minimum). There are two main such strategies: line search and trust region, which are illustrated in Fig. 3.6 and Fig. 3.7, respectively.

The line search approach consists of three main steps for every iteration:

1. Choose a suitable search direction (p_k in Fig. 3.6) from the current point x_k. 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 x_{k+1} and update all values.

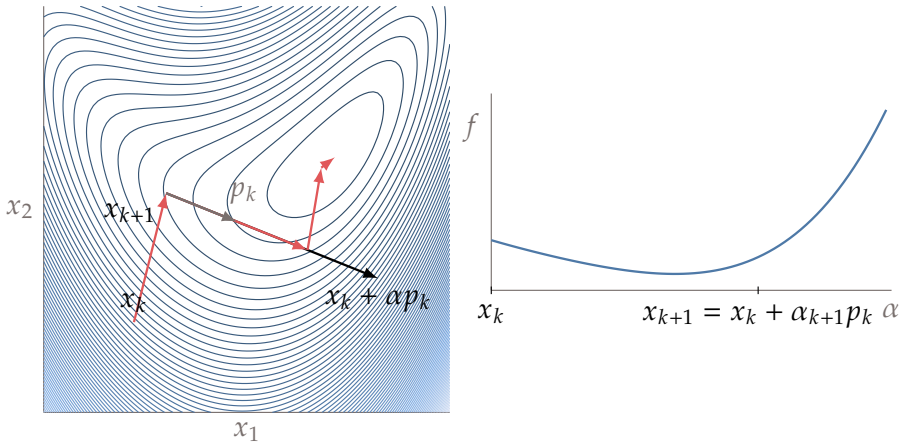


Figure 3.6: Line search approach for globalization.

Trust-region methods also consist of three steps:

1. Create a model about the current point, x_k. This model is based on a Taylor series approximation or can be another type of surrogate model.
2. Minimize the model within a *trust region* around the current point.

3. Move to the new point, update values, and adapt the size of the trust region.

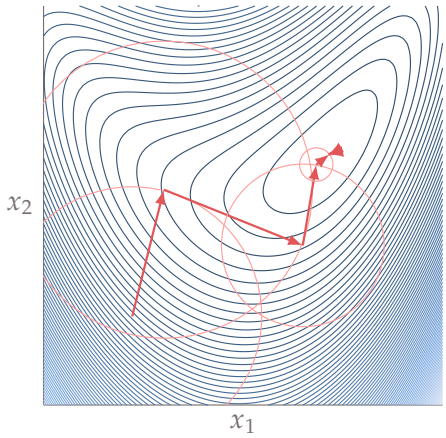


Figure 3.7: Trust region approach for globalization. In this case, where the trust regions are circular in this case.

Both of these are iterative processes that must be repeated until some convergence criterion is satisfied. The first step in both approaches corresponds to a *major* iteration, while the second step might require more function evaluations corresponding to *minor* iterations. We focus mostly on the line search strategies as they are more common, but we also introduce the trust-region procedure.

Before moving on it is worth briefly mentioning the relationship between optimization and root finding. Root finding could be posed as an optimization problem where we minimize $||f(x)||$. Conversely, optimization could be thought of root finding where we solve the problem $\nabla f = 0$. However, these are not the most efficient ways to approach these respective problems. Furthermore, solving $\nabla f = 0$ is not necessarily sufficient—it provides a stationary point and not necessarily a minimum. More information is need to ensure a minimum. For example, with a continuous one-dimensional function we can bracket a root with only two values, whereas to bracket a minimum we need three values. Still, because of the similarities, some algorithms have been adapted with variants for root finding and variants for optimization.

3.4 Line Search

As mentioned in the previous section, there are two main subproblems in line-search gradient-based optimization algorithms: choosing the search direction, and determining how far to go in that direction. Line search algorithms solve

the second subproblem. In the next section we introduce several methods for choosing the search direction. For now, we assume that for a given line search starting at x_k, we are given a suitable search direction p_k along which we are going to search. The line search methods we explore require that the search direction p_k be a *descent direction*, so that $\nabla f_k^T p_k < 0$ as previously shown in Fig. 3.4. This guarantees that f can be reduced by stepping some distance along this direction.

The movement in the search direction from one iteration to another can be written as

$$x_{k+1} = x_k + \alpha_k p_k,$$

where the scalar α_k expresses how far we go in the current direction p_k. The objective of the line search is to determine the magnitude of the scalar α_k, which in turn will determine 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 α_k. This two-step procedure is outlined in Algorithm 4 and applies to all gradient-based unconstrained optimization algorithms that use a line search.

Algorithm 4 A general gradient-based unconstrained optimization algorithm.

Input: Starting point x_0, convergence tolerance τ
  while $||\nabla f||_\infty > \tau$ **do**
    Determine search direction, p_k
    Determine step length, α_k
    Update design variables: $x_{k+1} = x_k + \alpha_k p_k$
    Increment iteration index $k = k + 1$
  end while
Return: Optimal point x^* and corresponding function value $f(x^*)$

The goal of the line search is not to find the value of α_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. Since the overall optimization needs to find a point in n-dimensional space, the search direction might change drastically between line searches, so spending too much computational effort on each line search is generally not worthwhile.

Consider the function shown in Fig. 3.8. 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.

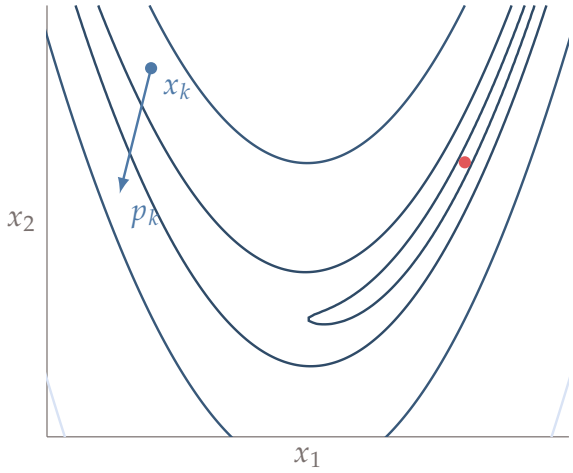


Figure 3.8: Contour plot of the Rosenbrock function with the location of the minimum denoted by the dot on the right side of the figure. A sample starting point x_k with direction vector p_k is overlaid.

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

$$\phi(\alpha) = f(x_k + \alpha p_k), \tag{3.19}$$

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

$$\phi'(\alpha) = \frac{\partial[f(x)]}{\partial \alpha} = \frac{\partial[f(x)]}{\partial x}\frac{\partial x}{\partial \alpha} = \nabla f(x)^T p_k = \nabla f(x_k + \alpha p_k)^T p_k, \tag{3.20}$$

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

$$\phi'(0) = \nabla f_k^T p_k. \tag{3.21}$$

3.4.1 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

$$\phi(\alpha) \leq \phi(0) + \mu_1 \alpha \phi'(0) \tag{3.22}$$

for a constant $0 < \mu_1 \leq 1$. The quantity $\alpha\phi'(0)$ represents the expected decrease of the function, assuming the function continued at the same slope. The multiplier μ_1 simply reflects the fact 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 α that satisfies this condition for a smooth function.

The concept is illustrated in Fig. 3.9. The black line is our function, but all we know is the function value and slope (blue line) on the left-hand side of the figure. If the function were to continue with the same trend it would follow the blue line. Because we don't want to waste a lot of function values trying to find the best step, we will accept a small fraction of the expected decrease (the red line). Anything below the line is an acceptable point using this criterion.

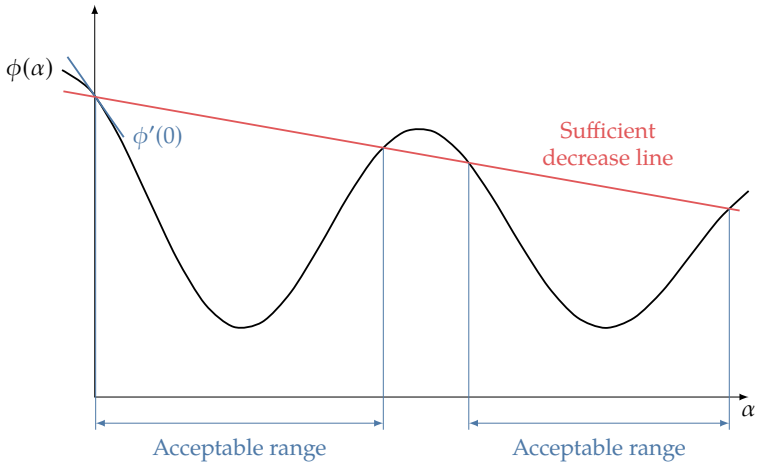


Figure 3.9: Sufficient decrease conditions.

Line search algorithms require a first guess for α. 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 α, 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 (Algorithm 5). This algorithm starts with a maximum step and successively reduces the step by a constant ratio ρ 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.

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 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 ρ, this scenario

Algorithm 5 Backtracking line search algorithm

Input: $\alpha > 0$ and $0 < \rho < 1$
  while $\phi(\alpha) > \phi(0) + \mu_1 \alpha \phi'(0)$ **do**
    $\alpha = \rho\alpha$
  end while
  $\alpha_k = \alpha$
Return: α_k

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 ρ. 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.

3.4.2 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 μ_1 in Eq. (3.22) is rather small. We could just increase μ_1 (that is, tilt the red line downward in Fig. 3.9) 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*:

$$|\phi'(\alpha)| \leq \mu_2 |\phi'(0)|. \tag{3.23}$$

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 μ_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 μ_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 μ_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 of 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 3.10 shows acceptable intervals that satisfy the strong Wolfe conditions.

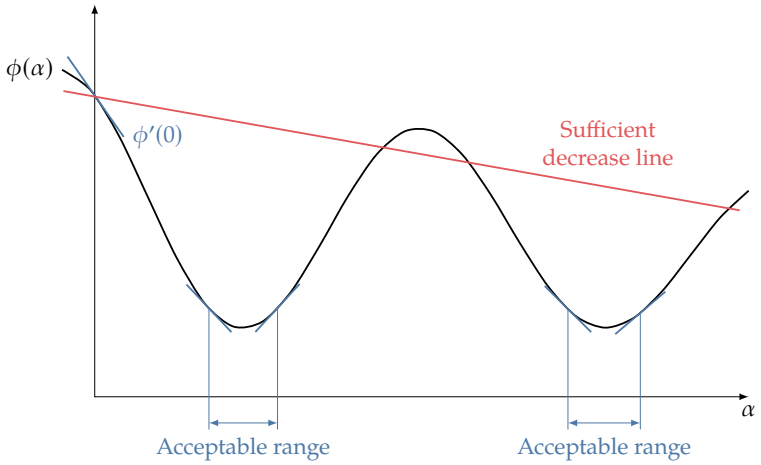


Figure 3.10: Steps that satisfy the strong Wolfe conditions

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 (ϕ'). 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 [1, Ch. 3]. 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.

The bracketing stage is detailed in Algorithm 6 and illustrated in Fig. 3.11; 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 6 Line search algorithm (bracketing stage)

Input: $\alpha_1 > 0$ ▷ Step size guess
  $\alpha_0 = 0$
  $i = 1$
  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**
      $\alpha^* =$ `pinpoint`(α_{i-1}, α_i) **return** α^*
    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**
      $\alpha^* =$ `pinpoint`(α_i, α_{i-1}) **return** α^*
    else
      Choose α_{i+1} such that $\alpha_i < \alpha_{i+1}$
    end if
    $i = i + 1$
  end while

The algorithm for the second stage, the `pinpoint`$(\alpha_{\text{low}}, \alpha_{\text{high}})$ function, is given in Algorithm 7. 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

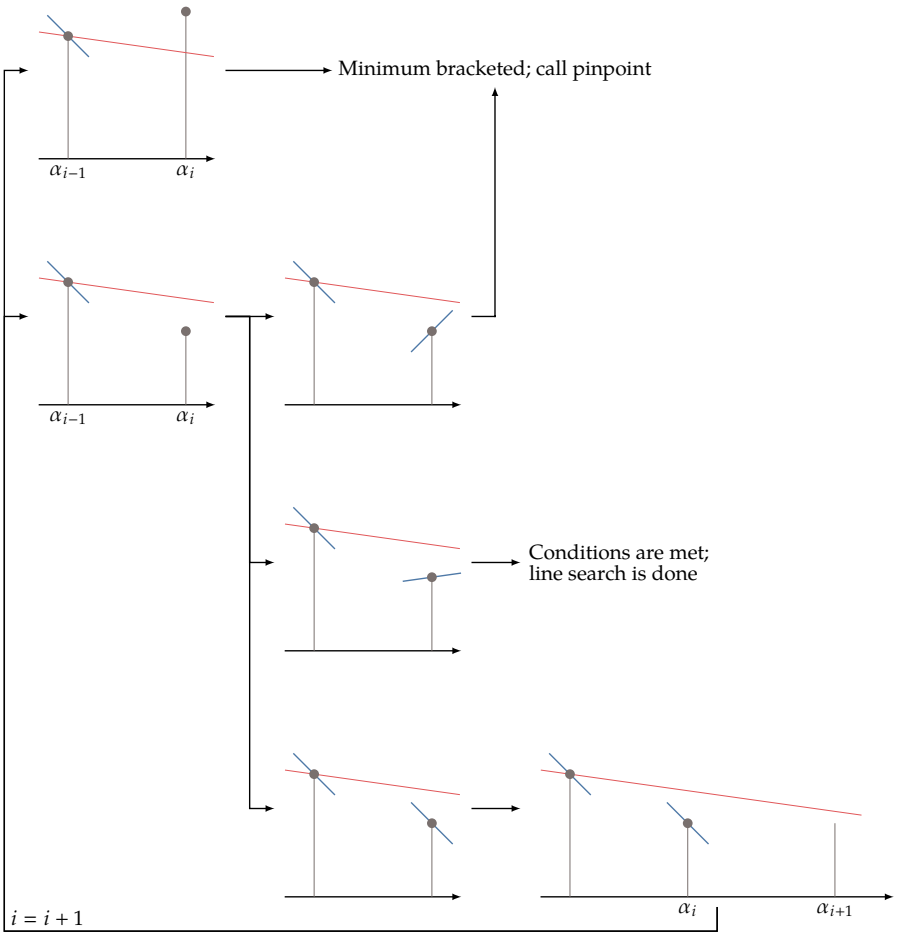


Figure 3.11: Visual representation of the bracketing algorithm

interpolation to estimate the minimum within the interval. If the two end points are α_1 and α_2, respectively, the minimum can be found analytically from the function values and derivative as

$$\alpha_{\min} = \frac{2\alpha_1 \left[\phi(\alpha_2) - \phi(\alpha_1)\right] + \phi'(\alpha_1)\left(\alpha_1^2 - \alpha_2^2\right)}{2\left[\phi(\alpha_2) - \phi(\alpha_1) + \phi'(\alpha_1)(\alpha_1 - \alpha_2)\right]}. \tag{3.24}$$

If we provide analytic gradients, or we already evaluated $\phi'(\alpha_i)$ (either as part of Algorithm 6 or as part of checking the strong Wolfe conditions in Algorithm 7), 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. 3.12. In the leftmost figure we construct a quadratic fit based on the function value and slope at α_{low} and the function value at α_{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 7 Pinpoint function for the line search algorithm

Input: $\alpha_{\text{low}}, \alpha_{\text{high}}$
  $j = 0$
  while true **do**
    Use quadratic (3.24) or cubic interpolation to find α_j in $[\alpha_{\text{low}}, \alpha_{\text{high}}]$
    Evaluate $\phi(\alpha_j)$
    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** α^*
      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

The line search defined by Algorithm 6 followed by Algorithm 7 is guaranteed to find a step length satisfying the strong Wolfe conditions for any parameters μ_1 and μ_2. A robust algorithm needs to consider additional issues.

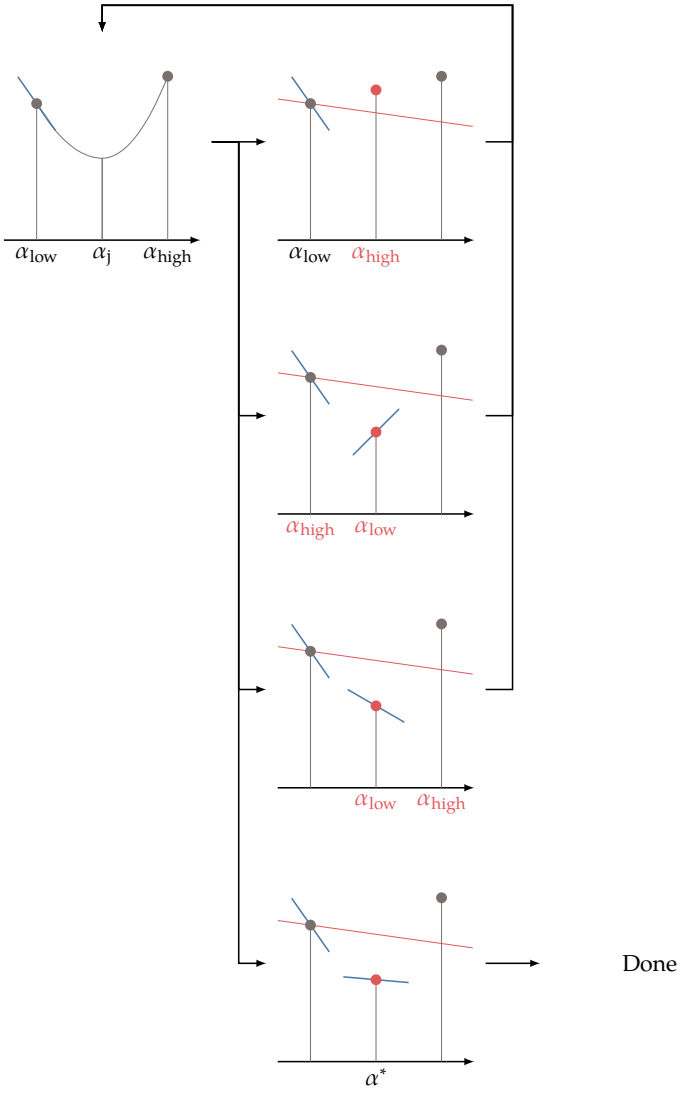


Figure 3.12: Visual representation of the pinpointing algorithm

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 3.1. How bracketing and pinpointing work together.

Consider the single-variable function $(x-3)x^3(x-6)^4$ plotted below.We perform a line search starting from $\alpha = 0$. The first guess in the line search is $\alpha = 1$, but this does not bracket a local minimum, so we try $\alpha = 2$, which does not bracket a minimum either. However, $\alpha = 4$ does bracket the minimum in the interval $[2, 4]$. In this case, the step lengths are chosen by doubling each successive guess, i.e., $\alpha_{i+1} = 2\alpha_i$.
Then, the pinpointing algorithm quickly converges to a point satisfying the strong Wolfe conditions close to $\alpha = 2$.

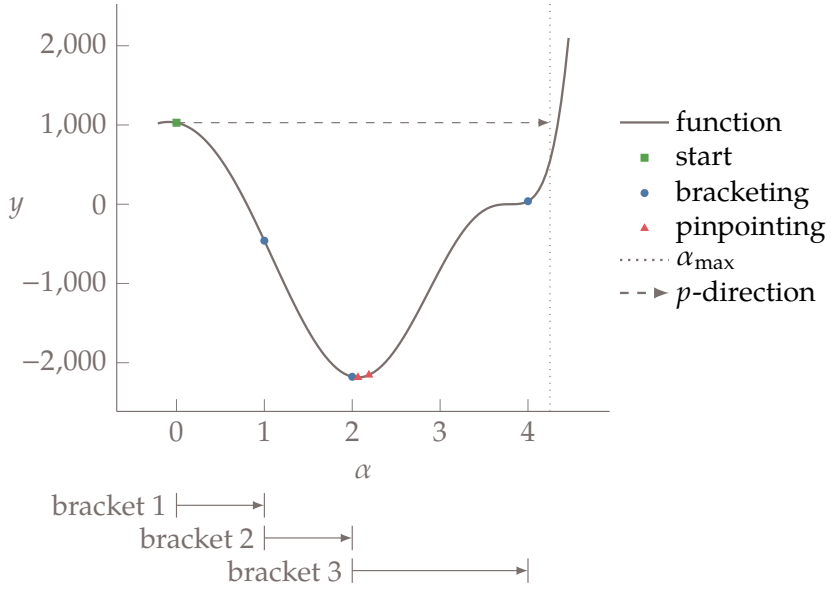


Figure 3.13: Example of a line search iteration.

3.5 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 (Algorithm 4). 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.

3.5.1 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 3.1, 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

$$p_k = -\nabla f_k. \tag{3.25}$$

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. Consider the quadratic shown in Fig. 3.14, which has very different curvatures in the x_1 and x_2 directions. The figure shows the iteration history produced by steepest descent on a simple quadratic function. We see that the path zigzags, and is inefficient in approaching the minimum.

This behavior is mathematically predictable, assuming we perform an exact line search at each iteration. An exact line search means selecting the optimal value for α along the line search:

$$\begin{aligned}
\frac{\partial f(x_k + \alpha p_k)}{\partial \alpha} &= 0 \\
\frac{\partial f(x_{k+1})}{\partial \alpha} &= 0 \\
\frac{\partial f(x_{k+1})}{\partial x_{k+1}} \frac{\partial (x_k + \alpha p_k)}{\partial \alpha} &= 0 \\
\nabla f_{k+1}^T p_k &= 0 \\
-p_{k+1}^T p_k &= 0
\end{aligned} \tag{3.26}$$

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.

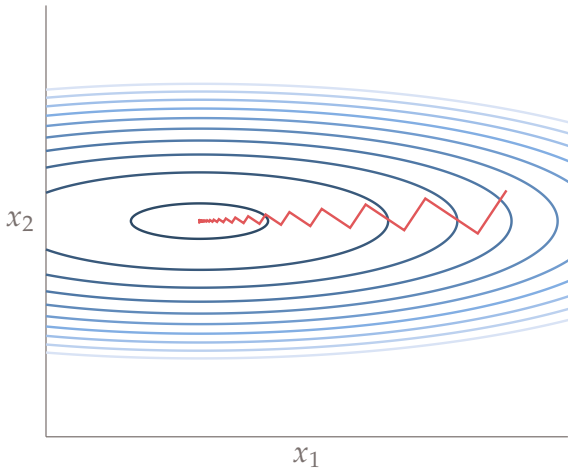


Figure 3.14: Iteration history for quadratic function $f = x_1^2 + 15x_2^2$, starting at $x_0 = (10, 1)$, using the steepest descent method with an exact line search. The total number of iterations was 111.

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

$$\alpha_k \nabla f_k^T p_k \approx \alpha_{k-1} \nabla f_{k-1}^T p_{k-1}. \tag{3.27}$$

Solving for the step length, and inserting the steepest descent direction, we get the guess

$$\alpha_k = \alpha_{k-1} \frac{\|\nabla f_{k-1}\|^2}{\|\nabla f_k\|^2}. \tag{3.28}$$

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.

Practical Tip 3.1. 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.

3.5.2 Conjugate Gradient

Steepest descent generally performs quite poorly, especially if the problem is not well scaled, like the quadratic example in Figure 3.14. 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. 3.15).

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

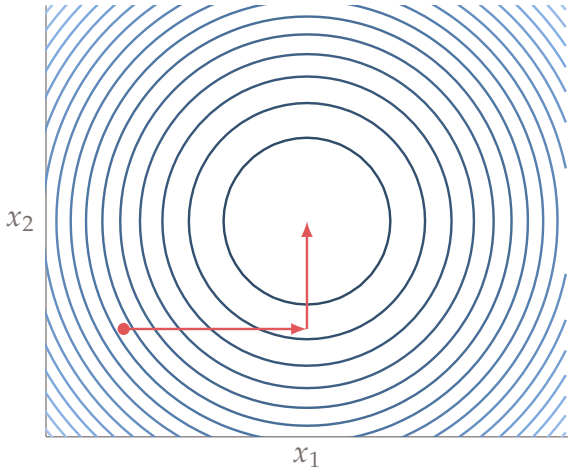


Figure 3.15: For a quadratic function with perfectly circular contours (condition number of 1) we can find the minimum in n steps, where n is the number of dimensions, by using a coordinate search. A coordinate search just means that we find that minimum in each coordinate sequentially. So in the above example we searched in the x_1 direction to find the minimum along that line, then searched in the x_2 direction.

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

$$p_i^T H p_j = 0 \text{ for all } i \neq j. \tag{3.29}$$

Note that 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. 3.16). That's a big improvement over steepest descent, which as we have seen can take many iterations to converge on a stretched 2-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

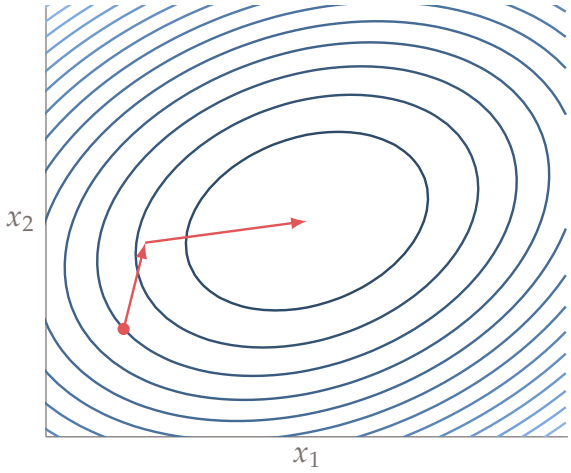


Figure 3.16: For any quadratic function we can find the minimum in n steps, where n is the number of dimensions, by searching along conjugate directions. Conjugate directions are "orthogonal" with respect to the Hessian.

on the search direction and gradient from the previous iteration:

$$p_k = -\nabla f_k + \beta_k p_{k-1}, \tag{3.30}$$

where

$$\beta_k = \frac{\nabla f_k^T \nabla f_k}{\nabla f_{k-1}^T \nabla f_{k-1}}. \tag{3.31}$$

The parameter β can be interpreted as a "damping parameter" that prevents each search direction from varying too much relative to the previous one. When the function steepens, the damping becomes larger, and vice versa.

When using the conjugate gradient method to minimize the quadratic function from the last section, only two iterations are required to converge (Fig. 3.17). This is because it is a quadratic function of two variables and we performed exact line searches. Convergence on a more general function is shown later in Section 3.5.5.

3.5.3 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:

$$f(x_k + s_k) = f_k + \nabla f_k^T s_k + \frac{1}{2} s_k^T H_k s_k + \cdots, \tag{3.32}$$

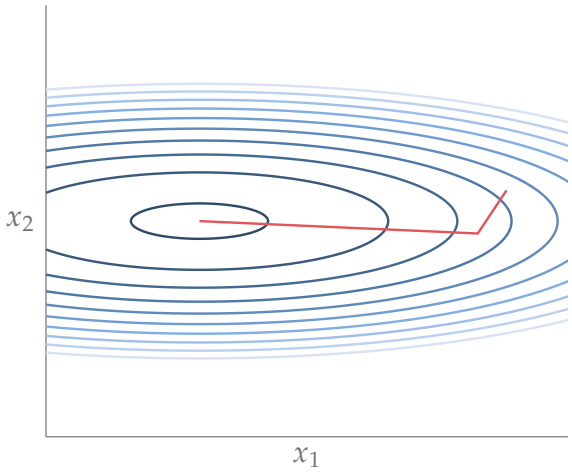


Figure 3.17: Iteration history for quadratic function $f = x_1^2 + 15x_2^2$, starting at $x_0 = (10, 1)$, using an exact line search and the conjugate gradient method. The total number of iterations is 2.

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:

$$\begin{aligned} \frac{\mathrm{d}f(x_k + s_k)}{\mathrm{d}s_k} &= \nabla f_k + H_k s_k = 0 \\ H_k s_k &= -\nabla f_k \\ s_k &= -H_k^{-1} \nabla f_k. \end{aligned} \tag{3.33}$$

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. 3.18). 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 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.

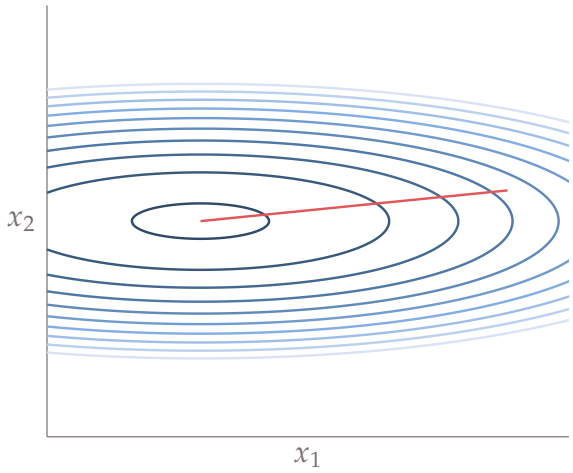


Figure 3.18: Iteration history for quadratic function $f = x_1^2 + 15x_2^2$, starting at $x_0 = (10, 1)$, using an exact line search and Newton's method. The total number of iterations is 1.

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: $f(x_k + s_k) > f(x_k)$.

   Solution: Rather than blindly accepting the new point $x_k + s_k$, we perform a line search starting with $p_k = s_k$ and $\alpha_k = 1$. Because the search direction s_k is a descent direction, if we backtrack 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).

3.5.4 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

$$f' \approx \frac{f_{k+1} - f_k}{x_{k+1} - x_k}, \tag{3.34}$$

where f' might be used to estimate the slope at k or $k+1$ (see Fig. 3.19). We can use a similar finite difference to estimate the curvature using the slopes and rearrange the equation to get

$$f''_{k+1}(x_{k+1} - x_k) = f'_{k+1} - f'_k. \tag{3.35}$$

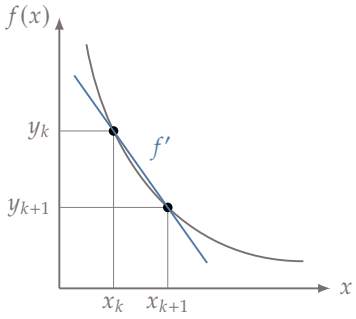


Figure 3.19: A tangent line at point x_k used to estimate f_{k+1}

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*

$$B_{k+1} s_k = \nabla f_{k+1} - \nabla f_k. \tag{3.36}$$

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 (3.36) is only possible if the predicted curvature is positive along the step, that is,

$$s_k^T \left(\nabla f_{k+1} - \nabla f_k \right) > 0. \tag{3.37}$$

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 [2, 3]. 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 [4, 5, 6, 7] 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:

$$p_k = -B_k^{-1} \nabla f_k. \tag{3.38}$$

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:

$$p_k = -V_k \nabla f_k. \tag{3.39}$$

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 (3.36) can be rewritten in terms of our new estimate V_{k+1} as:

$$V_{k+1}(\nabla f_{k+1} - \nabla f_k) = x_{k+1} - x_k. \tag{3.40}$$

Mathematically, the problem that we need to solve to estimate the next Hessian inverse V_{k+1} is:

$$\begin{aligned} \text{minimize} \quad & \|V_{k+1} - V_k\| \\ \text{with respect to} \quad & V_{k+1} \\ \text{subject to} \quad & V_{k+1} = V_{k+1}^T \\ & V_{k+1}(\nabla f_{k+1} - \nabla f_k) = x_{k+1} - x_k \end{aligned} \tag{3.41}$$

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 [8]. The solution for V_{k+1} is:

$$V_{k+1} = \left[I - \frac{s_k y_k^T}{s_k^T y_k} \right] V_k \left[I - \frac{y_k s_k^T}{s_k^T y_k} \right] + \frac{s_k s_k^T}{s_k^T y_k}. \tag{3.42}$$

where

$$s_k = x_{k+1} - x_k = \alpha_k p_k$$

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 the gradients at the end and start of the line search (the last to 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 numerator $s_k y_k T$ is an outer product that results in an $n_x \times n_x$ matrix, or rank 1. The division of this matrix by the scalar is performed element wise.

Eq. (3.42) 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. (3.39) results in

$$p_0 = -\nabla f_0 \tag{3.43}$$

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 (3.41) 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 identify matrix as suggested above, all subsequent updated produce positive definite matrices.

Figure 3.20 shows the iteration history on the same quadratic function from the previous sections, but with a BFGS Quasi-Newton method. Note that the first step is steepest descent, but the Hessian inverse update becomes accurate quickly.

3.5.5 Summary and Comparison of Search Direction Methods

All gradient-based methods presented so far follow Algorithm 4 and are characterized by how they choose the search direction p_k. The four methods we

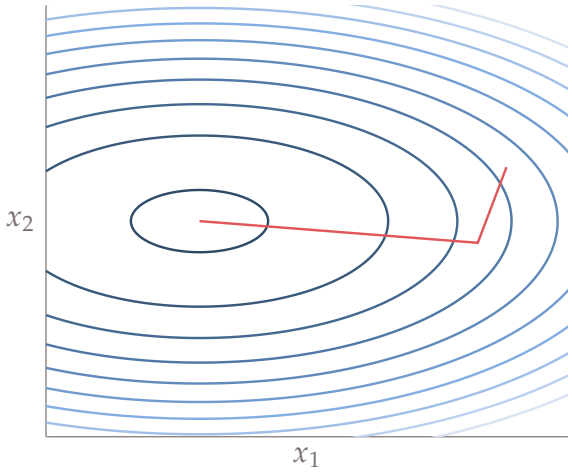


Figure 3.20: Iteration history for quadratic function $f = x_1^2 + 15x_2^2$, starting at $x_0 = (10, 1)$, using an exact line search and the BFGS Quasi-Newton method. The total number of iterations was 2.

have discussed yield the following search directions:

$$\begin{aligned} \text{Steepest descent:} \quad & p_k = -\nabla f_k \\ \text{Conjugate gradient:} \quad & p_k = -\nabla f_k + \beta_k p_{k-1} \\ \text{Newton:} \quad & p_k = -H_k^{-1} \nabla f_k \\ \text{Quasi-Newton:} \quad & p_k = -V_k \nabla f_k \end{aligned} \tag{3.44}$$

As an example, we apply all four methods on the Rosenbrock function:

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$$

The Rosenbrock function is a well-known starter optimization problem because it has only two dimensions, and so is easy to visualize, but has a long curved valley that makes it challenging to optimize. The convergence history for all four methods, from a common starting point, is shown in Fig. 3.22.

The differences in performance observed on this simple problem are fairly typical. Steepest descent does converge, but it takes a long time (12,050 iterations for this starting point). Conjugate gradient requires only a small change in how we compute the search direction, but yields a large improvement in performance (309 iterations). You can't tell from the figure, because the behavior is similar for the first few hundred iterations, but after that conjugate gradient starts converging superlinearly. Still, the methods that use second-order information are much more efficient than either of these first-order methods. In particular, the expected quadratic convergence behavior can be observed at

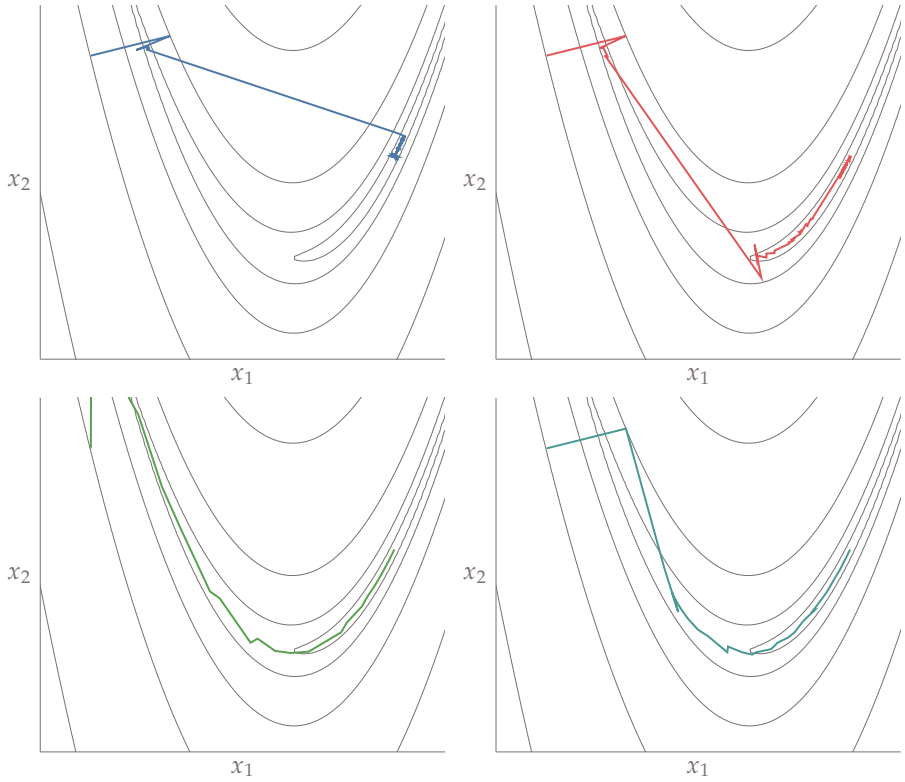


Figure 3.21: Optimization paths for the Rosenbrock function using steepest descent (top left), conjugate gradient (top right), Newton (bottom left), and BFGS (bottom right).

the end of both the Newton and Quasi-Newton curves. For this problem, the methods take 25 and 36 iterations to converge respectively.

It is important to note that 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 Newton's method requires a linear solve, 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.

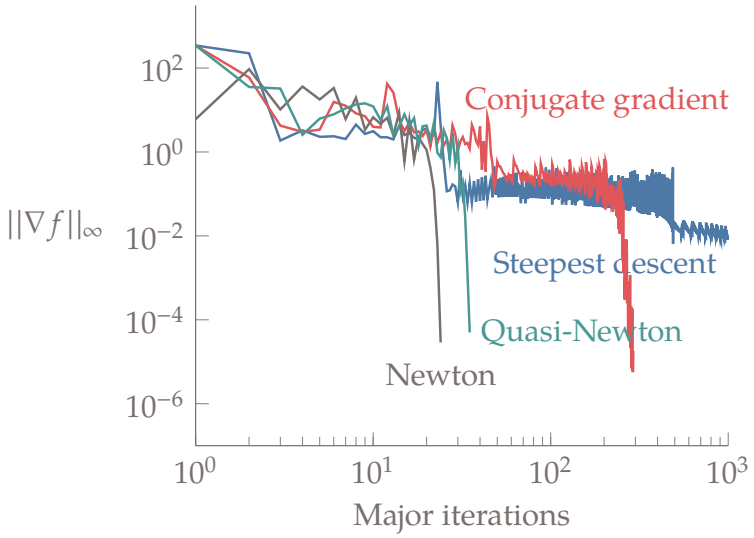


Figure 3.22: All four search direction methods are tested on the Rosenbrock function with a basic inexact line search and a starting point of $x_0 = (-2, 2)$. The Quasi-Newton method uses the BFGS update. The total number of function calls for the four methods, to a convergence tolerance of 10^{-6}, is 12050 for steepest descent, 309 for conjugate gradient, 25 for Newton, and 36 for quasi-Newton.

Practical Tip 3.2. Globalization strategies are effective for multi-modal problems.

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 globalization strategy. The simplest and most common approach is to use a *multistart* approach, meaning that we execute a gradient-based search multiple times, from different starting points. Starting points might be chosen from engineering intuition, randomly generated points, or sampling methods like Latin hypercube sampling (discussed in Chapter 12). 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, it may suggest that the design space wasn't 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 as discussed in Chapter 6. After some suitable initial exploration the designs in the population become starting points for gradient-based optimization. This type of approach allows for rigorous convergence that is typically not possible with a pure gradient-free approach.

3.6 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.

The trust-region subproblem is

$$\begin{aligned} \text{minimize} \quad & \tilde{f}(s) \\ \text{with respect to} \quad & s \\ \text{subject to} \quad & \|s\| \leq \Delta, \end{aligned} \tag{3.45}$$

where $\tilde{f}(s)$ is the local trust-region model, s is the step from the current iteration point, and Δ 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 ∞-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 [9]. 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.

Using a quadratic trust-region model and the Euclidean norm we can define the more specific subproblem

$$\begin{aligned} &\text{minimize} && \tilde{f}(s) = f_k + \nabla f_k^T s + \frac{1}{2} s^T B_k s \\ &\text{subject to} && \|s\|_2 \leq \Delta_k, \end{aligned} \tag{3.46}$$

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 9), 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 [9, 10, 1].

Example 3.2. Visualizing a trust region.

The left side of Fig. 3.23 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. (3.46)).

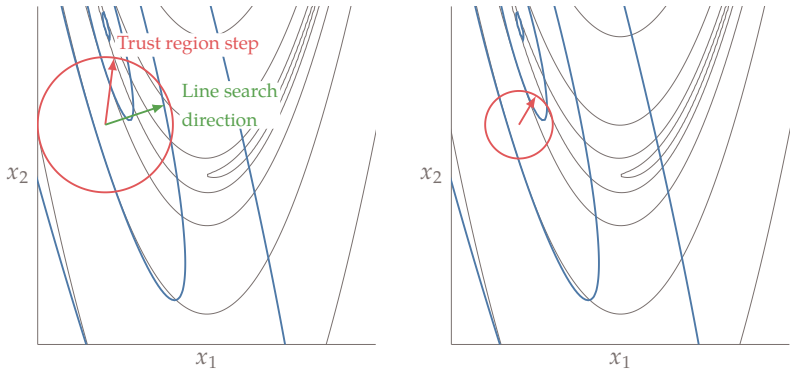


Figure 3.23: The black contour lines are the Rosenbrock function and the blue contours are a local quadratic approximation about the current iteration (where the arrow originate). The red circle represents a trust region. It is a safeguard to prevent steps beyond where the local model is likely to be valid. The trust-region step finds the minimum of the blue contours while remaining within the trust-region boundary.

3.6.1 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

$$r = \frac{f(x) - f(x+s)}{\tilde{f}(0) - \tilde{f}(s)}. \tag{3.47}$$

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 Algorithm 8 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 and Wright [1], but with recommended parameters from Conn, Gould, and Toint [9]. The initial value of Δ 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.

Algorithm 8 Trust-region algorithm

Input: Initial guess x_0, initial size of the trust region, Δ_0
  while not converged **do**
    Compute or estimate the Hessian
    Solve (approximately) Eq. (3.45) for s_k
    Compute r_k using Eq. (3.47)

    ▷ 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
Return: Optimum, x^*

3.6.2 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 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.

> **Practical Tip 3.3.** 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 4 is devoted to the subject of obtaining accurate derivatives.

3.7 Further Notes

- The first Wolfe condition (Eq. (3.22)) can be problematic near a local minimum because $\phi(0)$ and $\phi(\alpha)$ are very similar and so their subtraction is inaccurate. Hager and Zhang [11] introduced an approximate Wolfe condition with improved accuracy along with an efficient line search based on a secant method.
- A much more thorough introduction to the conjugate gradient method is given by Shewchuk [12].
- In the machine learning community the needs are sometimes quite different than most engineering design problems and different algorithms have emerged. For example, most machine learning methods are based on using large amounts of training data. The large amount of training data makes evaluating the gradient expensive. On the flip side, this structure can be exploited to provide reasonable estimates of the gradient by using only a subset of the training data. In the extreme case only one sample is used, but more commonly small groups of samples (called minibatches) are used, often generated stochastically. For example, if there were one million training samples then a single gradient evaluation would require evaluating all one million training samples. Alternatively, for a similar cost, a stochastic gradient descent method 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. Stochastic minibatching is easily applied to first-order methods and has thus driven innovation in alternative first-order methods like Momentum, Adam, and AMSGrad [13]. While some of these methods may seem rather ad hoc there is mathematical rigor to some of them [14]. 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.

- Conn et al. [9] provide 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.

Bibliography

[1] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, 2nd edition, 2006.

[2] William C. Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, 1(1):1–17, Feb 1991. ISSN 1095-7189. doi:10.1137/0801001.

[3] R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6(2):163–168, Aug 1963. ISSN 1460-2067. doi:10.1093/comjnl/6.2.163.

[4] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970. ISSN 1464-3634. doi:10.1093/imamat/6.1.76.

[5] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, Mar 1970. ISSN 1460-2067. doi:10.1093/comjnl/13.3.317.

[6] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–23, Jan 1970. ISSN 0025-5718. doi:10.1090/s0025-5718-1970-0258249-6.

[7] D. F. Shanno. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation*, 24(111):647–647, Sep 1970. ISSN 0025-5718. doi:10.1090/s0025-5718-1970-0274029-x.

[8] Roger Fletcher. *Practical Methods of Optimization*. Wiley, 2nd edition, 1987.

[9] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust Region Methods*. SIAM, Jan 2000. ISBN 0898714605.

[10] Trond Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, 20(3):626–637, Jun 1983. ISSN 1095-7170. doi:10.1137/0720042. URL http://dx.doi.org/10.1137/0720042.

[11] William W. Hager and Hongchao Zhang. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on Optimization*, 16(1):170–192, Jan 2005. ISSN 1095-7189. doi:10.1137/030601880.

[12] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburg, PA, Aug 1994.

[13] Sebastian Ruder. An overview of gradient descent optimization algorithms. arXiv:1609.04747, 2016.

[14] Gabriel Goh. Why momentum really works. *Distill*, 2017. doi:10.23915/distill.00006.

CHAPTER 4

Computing Derivatives

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.

4.1 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:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_{n_x}} \right]^T . \tag{4.1}$$

In general, however, optimization problems in engineering design are constrained. We will see in the next chapter, for constrained gradient-based optimization we also need 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 it as a vector-valued function, $f = [f_1, f_2, \ldots, f_{n_f}]^T$. The derivatives of all the functions of interest with respect to all the variables form the *Jacobian matrix*,

$$J(x) = \begin{bmatrix} \nabla {f_1}^T \\ \vdots \\ \nabla {f_{n_f}}^T \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_{n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{n_f}}{\partial x_1} & \cdots & \frac{\partial f_{n_f}}{\partial x_{n_x}} \end{bmatrix}, \tag{4.2}$$

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 this in index notation,

$$J_{ij} = \frac{\partial f_i}{\partial x_j}. \tag{4.3}$$

The gradient of function f_i shows up in row i of the Jacobian. Each column in the Jacobian is called the *tangent* with respect to a given variable x_j.

Example 4.1. Jacobian of a vector-valued function.

Consider the following function with two inputs and two outputs:

$$f(x) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1 x_2 + \sin x_1 \\ x_1 x_2 + x_2^2 \end{bmatrix}. \tag{4.4}$$

We can differentiate this symbolically to obtain exact reference values.

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \begin{bmatrix} x_2 + \cos x_1 & x_1 \\ x_2 & x_1 + 2x_2. \end{bmatrix} \tag{4.5}$$

We evaluate this at $x = (\pi/4, 2)$, which yields:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \begin{bmatrix} 2.707 & 0.785 \\ 2.000 & 4.785 \end{bmatrix}. \tag{4.6}$$

4.2 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. 4.1. 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 4.4). 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 4.6), as well as the complex-step method (Section 4.5). In the intermediate case we look at the model residuals and states (middle), which are the basis for the analytic methods (Section 4.7). 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 4.10).

Figure 4.1: Derivative computation methods can consider three different levels of information: function values, model states, and lines of code.

Practical Tip 4.1. Find out the level of numerical noise in you model.
Except for the simplest models, which can achieve machine zero noise, all models have some level of numerical noise above that. This can be problematic when computing derivatives and performing optimization, so it is important to know the level of numerical noise you are dealing with. Explore how the output functions change with respect to small changes in the design variables to quantify the level of numerical noise.

Practical Tip 4.2. Identifying and fixing sources of numerical noise.
There are several common sources of model numerical noise, some of which can be mitigated.

- Partially converged solvers: Many model implementations contain internal convergence loops or solvers. The solution for this is to be aware of how these are implemented and to reduce the convergence tolerance.
- File input and output: Many legacy codes are driven through 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. This allows for data to be passed in memory without the need for any files. Another solution is to change the output precision in the files. This latter option is simpler, but does not yield the speed advantage of eliminating the unnecessary time spent reading and writing files.

Practical Tip 4.3. Fixing discontinuous models.
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.

4.3 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 Example 4.1), many computational models are implicitly defined and involve iterative solvers (see Chapter 2). 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 4.2. Difficulties associated with symbolic differentiation.

Consider the function defined in Algorithm 9. A symbolic math toolbox can be used to find the closed-form expression for $\mathrm{d}f/\mathrm{d}x$ shown in Fig. 4.2, 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 of Algorithm 9. Therefore, this approach is intractable for models of even moderate complexity.

Algorithm 9: A simple example that demonstrates the difficulties of symbolic differentiation.

Input: x
  $f = x$
  for i = 1:20 **do**
    $f = \sin(x + f)$
  end for
Return: f

```
dfdx =
cos(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin
     (x + sin(x + sin(x + sin(x + sin(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(x + sin(x + sin(x + sin(x + sin(x + sin(x + sin(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(x + sin(x + sin(x +
      sin(x + sin(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(x + sin(x + sin(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(x + sin(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(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(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(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(x + sin(x + sin
     (x + sin(2*x))))))))))*(cos(x + sin(x + sin(x + sin(x + sin(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(x + sin(x + sin(2*x
     ))))))))))*(cos(x + sin(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(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) + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
     + 1) + 1) + 1) + 1) + 1)
```

Figure 4.2: Symbolic derivative of the simple function in Algorithm 9.

Nevertheless, symbolic differentiation is still useful to derive derivatives of simple explicit components within a larger model when using analytic methods. Furthermore, algorithm differentiation relies on symbolic differentiation to differentiate each line of code in the model.

4.4 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.

4.4.1 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^{th} direction,

$$f(x + h\hat{e}_j) = f(x) + h\frac{\partial f}{\partial x_j} + \frac{h^2}{2!}\frac{\partial^2 f}{\partial {x_j}^2} + \frac{h^3}{3!}\frac{\partial^3 f}{\partial {x_j}^3} + \ldots, \tag{4.7}$$

where $\hat{e}_j$ is the unit vector in the j^{th} direction. Solving the above for the first derivative we obtain the finite-difference formula,

$$\frac{\partial f}{\partial x_j} = \frac{f(x + h\hat{e}_j) - f(x)}{h} + O(h), \tag{4.8}$$

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,

$$\frac{\partial f}{\partial x_j} = \lim_{h \to 0} \frac{f(x + h\hat{e}_j) - f(x)}{h} \approx \frac{f(x + h\hat{e}_j) - f(x)}{h}. \tag{4.9}$$

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. 4.3.

The backward difference approximation can be obtained by replacing h with $-h$ to yield,

$$\frac{\partial f}{\partial x_j} = \frac{f(x) - f(x - h\hat{e}_j)}{h} + O(h), \tag{4.10}$$

which is also a first order approximation.

Assuming each function evaluation yields the full vector f, the above formulas compute the j^{th} column of the Jacobian (4.2). 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.

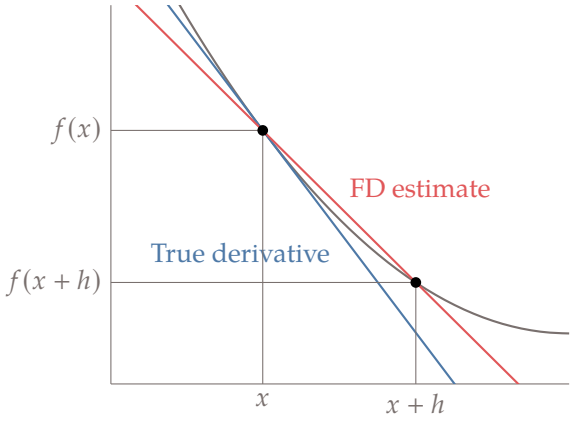


Figure 4.3: Exact derivative compare to a forward-difference finite-difference approximation.

For a second-order estimate of the first derivative, we can use the expansion of $f(x - h\hat{e}_j)$ to obtain,

$$f(x - h\hat{e}_j) = f(x) - h\frac{\partial f}{\partial x_j} + \frac{h^2}{2!}\frac{\partial^2 f}{\partial {x_j}^2} - \frac{h^3}{3!}\frac{\partial^3 f}{\partial {x_j}^3} + \dots. \tag{4.11}$$

Then, if we subtract this from the expansion (4.7) and solve the resulting equation for the derivative of f, we get the *central-difference* formula,

$$\frac{\partial f}{\partial x_j} = \frac{f(x + h\hat{e}_j) - f(x - h\hat{e}_j)}{2h} + O(h^2). \tag{4.12}$$

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 (4.12) to estimate the second derivative instead of the first,

$$\frac{\partial^2 f}{\partial {x_j}^2} = \frac{\left.\frac{\partial f}{\partial x_j}\right|_{x+h\hat{e}_j} - \left.\frac{\partial f}{\partial x_j}\right|_{x-h\hat{e}_j}}{2h} + O(h^2). \tag{4.13}$$

and use central difference again to estimate both $f'(x + h)$ and $f'(x - h)$ in the

above equation to obtain,

$$\frac{\partial^2 f}{\partial {x_j}^2} = \frac{f(x + 2h\hat{e}_j) - 2f(x) + f(x - 2h\hat{e}_j)}{4h^2} + O(h). \tag{4.14}$$

The finite-different 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 only need to step in the direction of interest, p. Using the forward difference, for example,

$$D_p f = \frac{f(x + hp) - f(x)}{h} + O(h). \tag{4.15}$$

One application of directional derivatives is to compute the slope in line searches.

4.4.2 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 4.1, 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.

$f(x + h)$	+1.2345678901234**31**
$f(x)$	+1.234567890123456
Δ f	−0.000000000000025

Table 4.1: 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 ϵ 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.

> **Practical Tip 4.4.** 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.

4.4.3 Practical Implementation

A procedure for computing a Jacobian using forward finite-differences is detailed in Algorithm 10. 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.

Algorithm 10 Forward finite-difference computation of the gradients of a vector-valued function $f(x)$.

Input: f, x
  $f_0 = f(x)$ ▷ Evaluate reference values.
  $n_f = \text{length}(f)$
  $n_x = \text{length}(x)$
  $J = \text{zeros}(n_f, n_x)$ ▷ Initialize Jacobian.
  $h = 10^{-6}$ ▷ Default relative step size. Could be an input parameter(s).
  for $j = 1 : n_x$ **do**
    $\Delta x = \max(h, h \cdot x[j])$ ▷ Scaled step size, but not smaller than h.
    $x[j]$ += Δx ▷ Modification made in place. Could copy vector instead.
    $f_+ = f(x)$ ▷ Evaluate function at perturbed point.
    $J[:, j] = (f_+ - f_0) \;/\; \Delta x$
    $x[j]$ −= Δx ▷ Do not forget to reset!
  end for
Return: J

Finite-different approximations with large enough steps can help smooth out numerical noise or discontinuities in the model, but it is better to address

these problems within the model whenever possible.

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 $p = [1, \ldots, 1]$)), 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.

4.5 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 [1] and Lyness and Moler [2], 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,

$$f(x + ih\hat{e}_j) = f(x) + ih\frac{\partial f}{\partial x_j} - \frac{h^2}{2}\frac{\partial^2 f}{\partial {x_j}^2} - i\frac{h^3}{6}\frac{\partial^3 f}{\partial {x_j}^3} + \ldots \tag{4.16}$$

Taking the imaginary parts of both sides of this equation,

$$\text{Im}[f(x + ih\hat{e}_j)] = h\frac{\partial f}{\partial x_j} - \frac{h^3}{6}\frac{\partial^3 f}{\partial {x_j}^3} + \ldots \tag{4.17}$$

Dividing this by h and solving for $\partial f / \partial x_j$ yields the *complex-step derivative approximation*,

$$\frac{\partial f}{\partial x_j} = \frac{\text{Im}\left[f(x + ih\hat{e}_j)\right]}{h} + O(h^2), \tag{4.18}$$

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 $f(x)$ 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. In Tip 4.6 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 (4.18) will match the precision of f. This is a tremendous advantage over the finite-difference approximations (4.8) and (4.12).

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 (4.16), we obtain the value of the function on the real axis,

$$f(x) = \text{Re}\left[f(x + ih\hat{e}_j)\right] + O(h^2). \tag{4.19}$$

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 (4.19) the truncation error is eliminated if h is such that

$$h^2 \left| \frac{1}{2} \frac{\partial^2 f}{\partial {x_j}^2} \right| < \epsilon \left| f(x) \right|, \tag{4.20}$$

where ϵ is the relative working precision of the function evaluation. Similarly, for the derivative approximation we require

$$h^2 \left| \frac{1}{6} \frac{\partial^3 f}{\partial {x_j}^3} \right| < \epsilon \left| f'(x) \right|. \tag{4.21}$$

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.

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 (Algorithm 11) 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 11 Procedure for computing the gradients of a vector-valued function $f(x)$ using the complex-step method.

Input: f, x
  n_x = length(x)
  n_f = length(f)
  J = zeros(n_f, n_x) ▷ Initialize Jacobian.
  $h = 10^{-40}$ ▷ Typical "small enough" step size.
  for $j = 1 : n_x$ **do**
    $x[j]$ += complex(0, h) ▷ Modify in place; could copy vector instead.
    $f_+ = f(x)$ ▷ Evaluate function perturbed with complex step.
    $J[:, j]$ = imag(f_+) / h ▷ Extract derivatives from imaginary part.
    $x[j]$ −= complex(0, h) ▷ Do not forget to reset!
  end for
Return: J

Practical Tip 4.5. Check the convergence of the imaginary part when dealing with iterative solvers.
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 4.3. Complex step accuracy compared to finite differences.

To show the how the complex-step method works, consider the following

analytic function:

$$f(x) = \frac{e^x}{\sqrt{\sin^3 x + \cos^3 x}}. \tag{4.22}$$

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

$$\epsilon = \frac{\left| \frac{\mathrm{d}f}{\mathrm{d}x}_{\text{approx}} - \frac{\mathrm{d}f}{\mathrm{d}x}_{\text{exact}} \right|}{\left| \frac{\mathrm{d}f}{\mathrm{d}x}_{\text{exact}} \right|} \tag{4.23}$$

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. 4.4. 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.

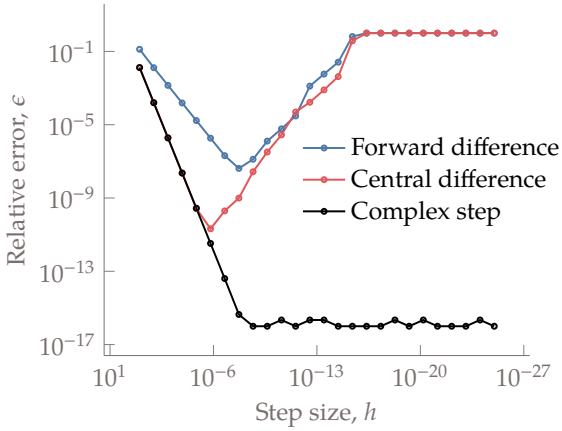


Figure 4.4: Relative error of the derivative approximations as the step size decreases. Finite-difference approximations initially converge as the truncation error decrease, but when the step is too small, the subtractive cancellation errors become overwhelming. The complex-step approximation does not 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.

Practical Tip 4.6. 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

$$|z| = \sqrt{x^2 + y^2}. \tag{4.24}$$

This definition is not complex analytic, which is required in the derivation of the complex-step derivative approximation. The following definition

of absolute value,

$$\text{abs}(x + iy) = \begin{cases} -x - iy, & \text{if } x < 0 \\ +x + iy, & \text{if } x \geq 0, \end{cases} \tag{4.25}$$

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. [3] provide more details on the problematic functions and how to implement the complex-step method in various programming languages.[a]

[a] https://goo.gl/TXqnpC

4.6 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 [4, 5]. 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.

4.6.1 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 $v = v_1, \ldots, v_i, \ldots, v_N$, 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:

$$v_i = V_i(v_1, v_2, \ldots, v_i, \ldots, v_n), \tag{4.26}$$

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

$$v_i = V_i(v_1, v_2, \ldots, v_{i-1}). \tag{4.27}$$

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.

4.6.2 Forward Mode AD

The chain rule for the forward mode can be written as:

$$\frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \sum_{k=j}^{i-1} \frac{\partial V_i}{\partial v_k} \frac{\mathrm{d}v_k}{\mathrm{d}v_j}, \tag{4.28}$$

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 (4.27), 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:

$$\dot{v}_i \equiv \frac{\mathrm{d}v_i}{\mathrm{d}v_j}. \tag{4.29}$$

Once we are done applying the chain rule (4.28) 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 $\mathrm{d}f/\mathrm{d}x$. 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

$$\begin{aligned}
\dot{v}_1 &= 1 \\
\dot{v}_2 &= \frac{\partial V_2}{\partial v_1}\dot{v}_1 \\
\dot{v}_3 &= \frac{\partial V_3}{\partial v_1}\dot{v}_1 + \frac{\partial V_3}{\partial v_2}\dot{v}_2 \\
\dot{v}_4 &= \frac{\partial V_4}{\partial v_1}\dot{v}_1 + \frac{\partial V_4}{\partial v_2}\dot{v}_2 + \frac{\partial V_4}{\partial v_3}\dot{v}_3 \equiv \frac{\mathrm{d}f}{\mathrm{d}x}
\end{aligned} \tag{4.30}$$

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., $\mathrm{d}v_3/\mathrm{d}v_1 \equiv \dot{v}_3$ in Eq. (4.30)) 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., $\mathrm{d}v_4/\mathrm{d}v_2$) 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 4.4. Forward mode AD for explicit functions.

Consider the function with two inputs and two outputs from Example 4.1. 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.

$$
\begin{aligned}
v_1 &= V_1(v_1) = x_1 \\
v_2 &= V_2(v_2) = x_2 \\
v_3 &= V_3(v_1, v_2) = v_1 v_2 \\
v_4 &= V_4(v_1) = \sin v_1 \\
v_5 &= V_5(v_3, v_4) = v_3 + v_4 \quad &= f_1 \\
v_6 &= V_6(v_2) = v_2^2 \\
v_7 &= V_7(v_3, v_6) = v_3 + v_6 \quad &= f_2
\end{aligned}
\tag{4.31}
$$

The operations above result in the dependency graph shown in Fig. 4.5.

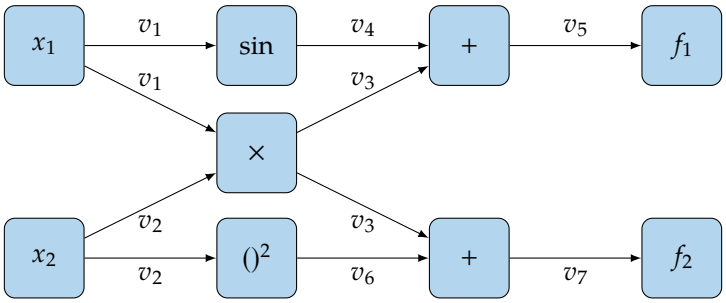


Figure 4.5: Dependency graph for the numerical example evaluations (4.31).

Say we want to compute $\mathrm{d}f_2/\mathrm{d}x_1$, which in our example is $\mathrm{d}v_7/\mathrm{d}v_1$. The evaluation point is the same as in Example 4.1: $x = (\pi/4, 2)$. 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,

$$
\begin{aligned}
\dot{v}_1 &= 1 \\
\dot{v}_2 &= 0 \\
\dot{v}_3 &= \frac{\partial V_3}{\partial v_1}\dot{v}_1 + \frac{\partial V_3}{\partial v_2}\dot{v}_2 \quad &= v_2\dot{v}_1 \quad &= 2 \\
\dot{v}_4 &= \frac{\partial V_4}{\partial v_1}\dot{v}_1 \quad &= \cos v_1 \dot{v}_1 \quad &= 0.707 \\
\dot{v}_5 &= \frac{\partial V_5}{\partial v_3}\dot{v}_3 + \frac{\partial V_5}{\partial v_4}\dot{v}_4 \quad &= \dot{v}_3 + \dot{v}_4 \quad &= 2.707 = \frac{\partial f_1}{\partial x_1} \\
\dot{v}_6 &= \frac{\partial V_6}{\partial v_2}\dot{v}_2 \quad &= 2v_2\dot{v}_2 \quad &= 0 \\
\dot{v}_7 &= \frac{\partial V_7}{\partial v_3}\dot{v}_3 + \frac{\partial V_7}{\partial v_6}\dot{v}_6 \quad &= \dot{v}_3 + \dot{v}_6 \quad &= 2 = \frac{\partial f_2}{\partial x_1},
\end{aligned}
\tag{4.32}
$$

> where we have evaluated the partial derivatives numerically. We now have a *procedure* (not a symbolic expression) for computing $\mathrm{d}f_2/\mathrm{d}x_1$ for any (x_1, x_2).
> While we set out to compute $\mathrm{d}f_2/\mathrm{d}x_1$, we also obtained $\mathrm{d}f_1/\mathrm{d}x_1$ 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, $\mathrm{d}f_1/\mathrm{d}x_2$, 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 $x \equiv [v_1, \ldots, v_{n_x}]^T$. To compute the Cartesian derivative with respect to x_1, for example, we would set the seed as a vector $\dot{v} = [1, 0, \ldots, 0]^T$, and similarly for the other components. To compute a directional derivative in direction p, we would set the seed to the vector $\dot{v} = p/||p||$.

4.6.3 Reverse Mode AD

The *reverse mode* is also based on the chain rule, but using the alternative form:

$$\frac{\mathrm{d}v_i}{\mathrm{d}v_j} = \sum_{k=j+1}^{i} \frac{\partial V_k}{\partial v_j} \frac{\mathrm{d}v_i}{\mathrm{d}v_k}. \tag{4.33}$$

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 (4.29), we define a more convenient notation for the variables that carry the total derivatives with a fixed i,

$$\bar{v}_j \equiv \frac{\mathrm{d}v_i}{\mathrm{d}v_j}, \tag{4.34}$$

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 (4.33) 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):

$$\begin{aligned}
\bar{v}_4 &= 1 \\
\bar{v}_3 &= \frac{\partial V_4}{\partial v_3}\bar{v}_4 \\
\bar{v}_2 &= \frac{\partial V_3}{\partial v_2}\bar{v}_3 + \frac{\partial V_4}{\partial v_2}\bar{v}_4 \\
\bar{v}_1 &= \frac{\partial V_2}{\partial v_1}\bar{v}_2 + \frac{\partial V_3}{\partial v_1}\bar{v}_3 + \frac{\partial V_4}{\partial v_1}\bar{v}_4 \equiv \frac{\mathrm{d}f}{\mathrm{d}x}
\end{aligned} \tag{4.35}$$

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., $\mathrm{d}v_4/\mathrm{d}v_2 \equiv \bar{v}_2$) or requires one more line of code to be evaluated. However, if we want the derivatives with respect to additional outputs (e.g., $\mathrm{d}v_3/dv_1$) 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 4.5. 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,

$$
\begin{aligned}
&\bar{v}_7 = 1 \\
&\bar{v}_6 = \frac{\partial V_7}{\partial v_6}\bar{v}_7 && = \bar{v}_7 && = 1 \\
&\bar{v}_5 = && = && = 0 \text{ (nothing depends on } v_5) \\
&\bar{v}_4 = \frac{\partial V_5}{\partial v_4}\bar{v}_5 && = \bar{v}_5 && = 0 \\
&\bar{v}_3 = \frac{\partial V_7}{\partial v_3}\bar{v}_7 + \frac{\partial V_5}{\partial v_3}\bar{v}_5 && = \bar{v}_7 + \bar{v}_5 && = 1 \\
&\bar{v}_2 = \frac{\partial V_6}{\partial v_2}\bar{v}_6 + \frac{\partial V_3}{\partial v_2}\bar{v}_3 && = 2v_2\bar{v}_6 + v_1\bar{v}_3 && = 4.785 = \frac{\partial f_2}{\partial x_2} \\
&\bar{v}_1 = \frac{\partial V_4}{\partial v_1}\bar{v}_4 + \frac{\partial V_3}{\partial v_1}\bar{v}_3 && = (\cos v_1)\bar{v}_4 + v_2\bar{v}_3 && = 2 = \frac{\partial f_2}{\partial x_1},
\end{aligned}
\tag{4.36}
$$

While we set out to evaluate $\mathrm{d}f_2/\mathrm{d}x_1$, we also computed $\mathrm{d}f_2/\mathrm{d}x_2$ 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, $f \equiv [v_{n-n_f}, \ldots, v_n]^T$. To find the derivatives of f_1 in a vectorized operation, we would set $\bar{v} = [1, 0, \ldots, 0]^T$. 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 particular 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 4.6. 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 in Algorithm 12, using a forward mode, for the algorithm we used earlier to illustrate the difficulties of symbolic differentiation (Algorithm 9). 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.

Algorithm 12: Forward mode automatic differentiation of Algorithm 9 using source code transformation.

Input: $x, \dot{x}$ ▷ Set seed $\dot{x} = 1$ to get $\mathrm{d}f/\mathrm{d}x$
  $f = x$
  $\dot{f} = \dot{x}$
  for i = 1:20 **do**
    $f = \sin(x + f)$
    $\dot{f} = (\dot{x} + \dot{f}) \cdot \cos(x + f)$
  end for
Return: $f, \dot{f}$ ▷ $\mathrm{d}f/\mathrm{d}x$ is given by $\dot{f}$

The reverse mode AD version of the same function is shown in Algorithm 13. 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).

Algorithm 13: Reverse mode automatic differentiation of Algorithm 9 using source code transformation.

Input: $x, \bar{f}$ ▷ Set $\bar{f} = 1$ to get $\mathrm{d}f/\mathrm{d}x$
  $f = x$
  for i = 1:20 **do**
    push(f) ▷ Put current value of f on top of stack
    $f = \sin(x + f)$
  end for
  for i = 20:1 **do** ▷ Do the reverse loop
    f = pop() ▷ Get value of f from top stack
    $\bar{f} = \cos(x + f) \cdot \bar{f}$
  end for
  $\bar{x} = \bar{x} + \bar{f}$
Return: $f, \bar{x}$ ▷ $\mathrm{d}f/\mathrm{d}x$ is given by $\bar{x}$

Practical Tip 4.7. 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 [6], C/C++, Python [7], Julia [8], and Matlab [9]. 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.
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.

4.7 Analytic Methods—Direct and Adjoint

Direct and adjoint methods—also known jointly as 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, 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 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.

4.7.1 Residuals and Functions

As mentioned in Chapter 2, a numerical model can be written as a set of residuals that need to be driven to zero,

$$r = R(x, u(x)) = 0, \tag{4.37}$$

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.,

$$f = F(x, u(x)). \tag{4.38}$$

Therefore, f depends not only explicitly on the design variables, but also implicitly through the governing equations (4.37). This dependency is illustrated in Fig. 4.6.

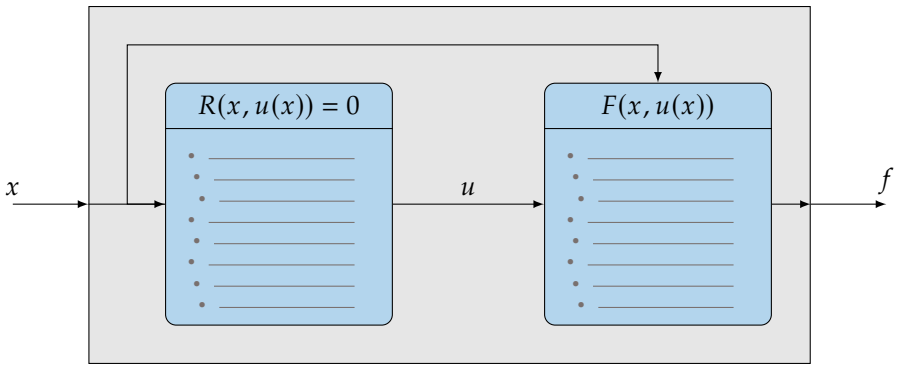


Figure 4.6: Relationship between functions and design variables for a general set of implicit equations. The implicit equations $r = 0$ define the states u for a given x, so the n_f functions of interest f end up depending both explicitly and implicitly on the n_x design variables x.

4.7.2 Direct and Adjoint Derivative Equations

The derivatives we ultimately want to compute are the ones in the Jacobian $\mathrm{d}f/\mathrm{d}x$. Given the explicit and implicit dependence of f on x, we can use chain rule to write the total derivative Jacobian of f as

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}x} \tag{4.39}$$

where the result is an $(n_f \times n_x)$ matrix.

In this context, the total derivatives $\mathrm{d}f/\mathrm{d}x$ take into account the change in u that is required to keep the residuals of the governing equations Eq. (4.37) 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. 4.6. 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, $\mathrm{d}u/\mathrm{d}x$ 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 (4.37) 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:

$$\frac{\mathrm{d}r}{\mathrm{d}x} = \frac{\partial R}{\partial x} + \frac{\partial R}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}x} = 0, \tag{4.40}$$

where $\mathrm{d}r/\mathrm{d}x$ and $\mathrm{d}u/\mathrm{d}x$ are both $(n_y \times n_x)$ matrices, and the Jacobian $\partial R/\partial u$ is a square matrix of size $(n_y \times n_y)$.

We can visualize the requirement for the total derivative (4.40) to be zero in Fig. 4.7, 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 $\mathrm{d}x$ must be accompanied by the appropriate change $\mathrm{d}y$ 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 (4.40) with respect to x must be zero.

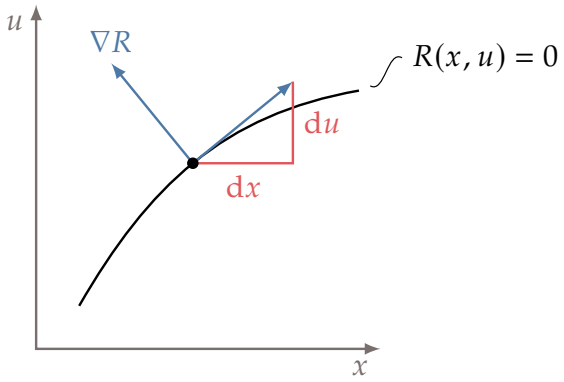


Figure 4.7: The residuals of the governing equations determine the values of u for a given x. Given a point that satisfies the equations, a perturbation in x about that point must be accompanied by the appropriate change in u for the equations to remain satisfied.

As in the total derivative of the function of interest (4.39), the partial deriva-

tives 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.,

$$\frac{\mathrm{d}u}{\mathrm{d}x} = -\left[\frac{\partial R}{\partial u}\right]^{-1} \frac{\partial R}{\partial x}, \tag{4.41}$$

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 $\mathrm{d}u/\mathrm{d}x$ 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 (4.39), we obtain:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial u}\left[\frac{\partial R}{\partial u}\right]^{-1} \frac{\partial R}{\partial x}, \tag{4.42}$$

where all the derivatives terms in the right-hand side are partial derivatives. The partial derivatives in this equation can be computed using any of the methods that we have described earlier: symbolic differentiation, finite difference, complex step, or AD.

The total derivative equation (4.42) 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 (4.41) and substituting $\mathrm{d}u/\mathrm{d}x$ into Eq. (4.39). We can express this by replacing the last two Jacobians with

$$\phi \equiv \left[\frac{\partial R}{\partial u}\right]^{-1} \frac{\partial R}{\partial x}. \tag{4.43}$$

Multiplying this by $\partial R/\partial u$, we get the linear system,

$$\frac{\partial R}{\partial u}\phi = \frac{\partial R}{\partial x}, \tag{4.44}$$

which we can solve to compute ϕ. Then, we can replace ϕ in the total derivative equation,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \frac{\partial F}{\partial u}\phi. \tag{4.45}$$

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 (4.44) 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 (4.42). 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,

$$\psi^T \equiv \frac{\partial F}{\partial u}\left[\frac{\partial R}{\partial u}\right]^{-1}, \tag{4.46}$$

where the columns of ψ, 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*,

$$\left[\frac{\partial R}{\partial u}\right]^T \psi = \left[\frac{\partial F}{\partial u}\right]^T. \tag{4.47}$$

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

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \psi^T \frac{\partial R}{\partial x}. \tag{4.48}$$

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, $\partial \mathcal{R}/\partial u$. 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 4.2. The direct method requires a solution of the linear system (4.44) for each design variable in x, while the adjoint method requires a solution of the linear system (4.47) for each function of interest in f.

Table 4.2: 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 4.7. Differentiating an implicit function.

Consider the following simplified equation for the natural frequency of a beam:

$$f = \lambda m^2 \tag{4.49}$$

where λ is a function of m through the following relationship

$$\frac{\lambda}{m} + \cos\lambda = 0 \tag{4.50}$$

Our goal is to compute the derivative $\mathrm{d}f/\mathrm{d}m$, but λ is an implicit function of m. In other words, we cannot find an explicit expression for λ as a function of m, substitute that expression into Eq. (4.49), and then differentiate normally.
Fortunately, the direct and adjoint methods will allow us to compute this derivative.
Referring back to our nomenclature,

$$\begin{aligned} F(x, u(x)) &\equiv F(m, \lambda(m)) = \lambda m^2, \\ R(x, u(x)) &\equiv R(m, \lambda(m)) = \frac{\lambda}{m} + \cos\lambda = 0 \end{aligned} \tag{4.51}$$

where m is the design variable and λ is the state variable. The partial derivatives that we need to compute the total derivative (4.42) are:

$$\begin{aligned} \frac{\partial F}{\partial x} &= \frac{\partial f}{\partial m} = 2\lambda m, & \frac{\partial F}{\partial u} &= \frac{\partial f}{\partial \lambda} = m^2 \\ \frac{\partial R}{\partial x} &= \frac{\partial R}{\partial m} = -\frac{\lambda}{m^2}, & \frac{\partial R}{\partial u} &= \frac{\partial R}{\partial \lambda} = \frac{1}{m} - \sin\lambda \end{aligned} \tag{4.52}$$

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 (4.42) yields:

$$\frac{\mathrm{d}f}{\mathrm{d}m} = 2\lambda m + \frac{\lambda}{\frac{1}{m} - \sin\lambda}. \tag{4.53}$$

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 4.8. 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:

$$\mathcal{R}_k(x, u) = N_k(x, u) - F_k = 0 \tag{4.54}$$

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:

$$\mathcal{R}_k(x, u) = K_{kj}(x)u_j - F_k = 0 \tag{4.55}$$

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:

$$\begin{aligned} \frac{\partial \mathcal{R}_k}{\partial x_i} &= \left[\frac{\partial K_{kj}}{\partial x_i}\right] u_j \\ \frac{\partial \mathcal{R}_k}{\partial u_j} &= K_{kj}. \end{aligned} \tag{4.56}$$

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 function of interest is the stress, which under the elastic assumption, is a linear function of the deflections that can be expressed as:

$$f_n(u) = \sigma_n(u) = S_{nj} u_j \tag{4.57}$$

Taking the partial derivatives,

$$\begin{aligned} \frac{\partial f_n}{\partial x_i} &= 0, \\ \frac{\partial f_n}{\partial u_j} &= S_{nj}. \end{aligned} \tag{4.58}$$

All of the partial derivatives above, except for one, require data that we already know. Putting everything together using the total derivative equation (4.42) yields,

$$\frac{\mathrm{d}\sigma_n}{\mathrm{d}x_i} = -S_{nj} K_{kj}^{-1} \left[\frac{\partial K_{kj}}{\partial x_i} \right] u_j \tag{4.59}$$

We can solve this either using the direct or adjoint method. The direct method would solve the linear system for ϕ, for one input i at a time, and then multiply through in the second equation:

$$\begin{aligned} K_{kj} \phi_j &= - \left[\frac{\partial K_{kj}}{\partial x_i} \right] u_j \\ \frac{\mathrm{d}\sigma_n}{\mathrm{d}x_i} &= S_{nj} \phi_j \end{aligned} \tag{4.60}$$

This approach is preferable if we have few design variables x_i and many stresses σ_n. Or, we can solve this with the adjoint approach. The adjoint method solves the linear system for ψ, one output n at a time, and then multiplies through in the second equation:

$$\begin{aligned} K_{kj} \psi_k &= -S_{nj} \\ \frac{\mathrm{d}\sigma_n}{\mathrm{d}x_i} &= \psi_k \left[\frac{\partial K_{kj}}{\partial x_i} \right] u_j. \end{aligned} \tag{4.61}$$

This approach is preferable when there are many design variables and few stresses.

4.8 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:

$$[x_1,\ x_2,\ \ldots,\ x_j + h,\ \ldots,\ x_{n_x}]^T \tag{4.62}$$

For many sparsity patterns, we can reduce computational cost greatly. As a motivating example, consider a square diagonal Jacobian:

$$\begin{bmatrix} J_{11} & 0 & 0 & 0 & 0 \\ 0 & J_{22} & 0 & 0 & 0 \\ 0 & 0 & J_{33} & 0 & 0 \\ 0 & 0 & 0 & J_{44} & 0 \\ 0 & 0 & 0 & 0 & J_{55} \end{bmatrix} \tag{4.63}$$

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 output 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:

$$[x_1 + h,\ x_2 + h,\ \ldots,\ x_j + h,\ \ldots,\ x_{n_x} + h]^T \tag{4.64}$$

and compute all the nonzero entires in one shot.

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 x 6 matrix as an example:

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \tag{4.65}$$

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:

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \Rightarrow \begin{bmatrix} J_{11} & J_{14} & J_{16} \\ J_{23} & J_{24} & 0 \\ J_{31} & J_{32} & 0 \\ 0 & J_{45} & 0 \\ J_{53} & J_{55} & J_{56} \end{bmatrix} \tag{4.66}$$

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.

$$\begin{bmatrix} J_{11} & 0 & 0 & J_{14} & 0 & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & J_{45} & 0 \\ 0 & 0 & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \Rightarrow \begin{bmatrix} J_{11} & 0 & 0 & J_{14} & J_{45} & J_{16} \\ 0 & 0 & J_{23} & J_{24} & 0 & 0 \\ J_{31} & J_{32} & J_{53} & 0 & J_{55} & J_{56} \end{bmatrix} \tag{4.67}$$

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 4.6.

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.

4.9 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 (4.26) considered when introducing AD:

$$v_i = V_i(v_1, v_2, \ldots, v_i, \ldots, v_n). \tag{4.68}$$

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,

$$r = R(v) = 0. \tag{4.69}$$

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 [10]:

$$\frac{\partial R}{\partial v}\frac{\mathrm{d}v}{\mathrm{d}r} = I = \frac{\partial R}{\partial v}^T \frac{\mathrm{d}v}{\mathrm{d}r}^T, \tag{4.70}$$

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, $\mathrm{d}f/\mathrm{d}x$ are a subset of $\mathrm{d}v/\mathrm{d}r$. 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 4.3), we can recover all the derivative computation methods using the UDE (4.70). The left-hand side represents forward (or direct) derivative computation, while the right-hand side represent reverse (or adjoint) derivative computation. 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.

Using the variable and residuals definitions from Table 4.3 for the monolithic method in the left hand side of the UDE (4.70), we get

$$\begin{bmatrix} I & 0 \\ -\dfrac{\partial F}{\partial x} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ \dfrac{\mathrm{d}f}{\mathrm{d}x} & I \end{bmatrix} = I, \tag{4.71}$$

Table 4.3: Variable and residual definition needed to recover the various derivative computation methods with the UDE (4.70). 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	$v - V(x,v)$

which yields the obvious result $\mathrm{d}f/\mathrm{d}x = \partial F/\partial x$. 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,

$$\begin{bmatrix} I & 0 & 0 \\ -\dfrac{\partial R_g}{\partial x} & -\dfrac{\partial R_g}{\partial u} & 0 \\ -\dfrac{\partial F}{\partial x} & -\dfrac{\partial F}{\partial u} & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ \dfrac{\mathrm{d}u}{\mathrm{d}x} & \dfrac{\mathrm{d}u}{\mathrm{d}r} & 0 \\ \dfrac{\mathrm{d}f}{\mathrm{d}x} & \dfrac{\mathrm{d}f}{\mathrm{d}r} & I \end{bmatrix} = I. \tag{4.72}$$

Since we are only interested in the $\mathrm{d}f/\mathrm{d}x$ 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 $\phi \equiv \mathrm{d}u/\mathrm{d}x$, we get the direct linear system (4.44) and the total derivative equation (4.45).

The right-hand side of the UDE yields the transposed system,

$$\begin{bmatrix} I & -\dfrac{\partial R_g}{\partial x} & -\dfrac{\partial F}{\partial x} \\ 0 & -\dfrac{\partial R_g}{\partial u} & -\dfrac{\partial F}{\partial u} \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} I & \dfrac{\mathrm{d}u}{\mathrm{d}x} & \dfrac{\mathrm{d}f}{\mathrm{d}x} \\ 0 & \dfrac{\mathrm{d}u}{\mathrm{d}r} & \dfrac{\mathrm{d}f}{\mathrm{d}r} \\ 0 & 0 & I \end{bmatrix} = I. \tag{4.73}$$

Similarly to the forward case, we ignore the block columns of the matrix of unknowns to focus on the block column containing $\mathrm{d}f/\mathrm{d}x$. Multiplying this out, and defining $\psi \equiv \mathrm{d}f/\mathrm{d}r$, we get the adjoint linear system (4.47) and the

corresponding total derivative equation (4.48). The definition of ψ 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

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ -\dfrac{\partial V_2}{\partial v_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ -\dfrac{\partial V_n}{\partial v_1} & \dots & -\dfrac{\partial V_n}{\partial v_{n-1}} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ \dfrac{\mathrm{d}v_2}{\mathrm{d}v_1} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ \dfrac{\mathrm{d}v_n}{\mathrm{d}v_1} & \dots & \dfrac{\mathrm{d}v_n}{\mathrm{d}v_{n-1}} & 1 \end{bmatrix} = I, \tag{4.74}$$

where the Jacobian $\mathrm{d}f/\mathrm{d}x$ is composed of a subset of derivatives in the corners near the $\mathrm{d}v_n/\mathrm{d}v_1$ 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

$$\begin{bmatrix} 1 & -\dfrac{\partial V_2}{\partial v_1} & \dots & -\dfrac{\partial V_n}{\partial v_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & -\dfrac{\partial V_n}{\partial v_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \dfrac{\mathrm{d}v_2}{\mathrm{d}v_1} & \dots & \dfrac{\mathrm{d}v_n}{\mathrm{d}v_1} \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & 1 & \dfrac{\mathrm{d}v_n}{\mathrm{d}v_{n-1}} \\ 0 & \dots & 0 & 1 \end{bmatrix} = I, \tag{4.75}$$

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 (4.70) have been used to develop a general framework for solving models and computing their derivatives [11, 12].

4.10 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 4 can be used, but some require modifications. The difference is that in MDO the computation of the functions of interest (objective and constraints) require 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 4.4, 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 4.7 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 (4.44)

is

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial u_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial u_1} & \cdots & \frac{\partial R_n}{\partial u_n} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} \frac{\partial R_1}{\partial x} \\ \vdots \\ \frac{\partial R_n}{\partial x} \end{bmatrix}, \tag{4.76}$$

where ϕ_i is the derivatives of the states from component i with respect to the design variable. Once we have solved for ϕ, we can use the coupled equivalent of the total derivative equation (4.45) to compute the derivatives.

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \begin{bmatrix} \frac{\partial F}{\partial u_1}, \ldots, \frac{\partial F}{\partial u_n} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_n \end{bmatrix}. \tag{4.77}$$

The coupled adjoint equations can be written as

$$\begin{bmatrix} \frac{\partial R_1}{\partial u_1}^T & \cdots & \frac{\partial R_1}{\partial u_n}^T \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial u_1}^T & \cdots & \frac{\partial R_n}{\partial u_n}^T \end{bmatrix}^T \begin{bmatrix} \psi_1 \\ \vdots \\ \psi_n \end{bmatrix} = \begin{bmatrix} \frac{\partial F}{\partial u_1} \\ \vdots \\ \frac{\partial F}{\partial u_n} \end{bmatrix}^T. \tag{4.78}$$

After solving for the coupled-adjoint vector using the equation above, we can use the total derivative equation to compute the desired derivatives,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \begin{bmatrix} \psi_1^T, \ldots, \psi_n^T \end{bmatrix} \begin{bmatrix} \frac{\partial R_1}{\partial x} \\ \vdots \\ \frac{\partial R_n}{\partial x} \end{bmatrix}. \tag{4.79}$$

Example 4.9. Coupled direct and coupled adjoint applied to two-equation system (residual form).

There is an alternative form for the coupled direct and adjoint methods that was not possible 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 2.4.2, 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 derive by defining the residuals as $R(y) \equiv y - Y(y) = 0$, where the states are now the coupling variables. The linear system for the direct method (4.76) then yields

$$\begin{bmatrix} I & -\frac{\partial Y_1}{\partial y_2} & \cdots & -\frac{\partial Y_1}{\partial y_n} \\ \vdots & \ddots & \ddots & \vdots \\ -\frac{\partial Y_n}{\partial y_1} & -\frac{\partial Y_n}{\partial y_2} & \cdots & I \end{bmatrix} \begin{bmatrix} \bar{\phi}_1 \\ \vdots \\ \bar{\phi}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial Y_1}{\partial x} \\ \vdots \\ \frac{\partial Y_n}{\partial x} \end{bmatrix}, \tag{4.80}$$

where $\bar{\phi}_i = \mathrm{d}y_i/\mathrm{d}x$. The total derivatives of the function of interest can then be computed with

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \begin{bmatrix} \frac{\partial F}{\partial y_1}, \dots, \frac{\partial F}{\partial y_n} \end{bmatrix} \begin{bmatrix} \bar{\phi}_1 \\ \vdots \\ \bar{\phi}_n \end{bmatrix}. \tag{4.81}$$

The functional form of the coupled adjoint equations can be similarly derived, yielding

$$\begin{bmatrix} I & -\frac{\partial Y_1}{\partial y_2}^T & \cdots & -\frac{\partial Y_1}{\partial y_n}^T \\ \vdots & \ddots & \ddots & \vdots \\ -\frac{\partial Y_n}{\partial y_1}^T & -\frac{\partial Y_n}{\partial y_2}^T & \cdots & I \end{bmatrix}^T \begin{bmatrix} \bar{\psi}_1 \\ \vdots \\ \bar{\psi}_n \end{bmatrix} = \begin{bmatrix} \frac{\partial Y_1}{\partial x} \\ \vdots \\ \frac{\partial Y_n}{\partial x} \end{bmatrix}, \tag{4.82}$$

After solving for the coupled-adjoint vector using the equation above, we can use the total derivative equation to compute the desired derivatives,

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\partial F}{\partial x} - \begin{bmatrix} \bar{\psi}_1^T, \dots, \bar{\psi}_n^T \end{bmatrix} \begin{bmatrix} \frac{\partial F}{\partial y_1} \\ \vdots \\ \frac{\partial F}{\partial y_n} \end{bmatrix}. \tag{4.83}$$

Example 4.10. Coupled direct and coupled adjoint applied to two-equation system (residual form).

Finally, the unification of the methods for computing derivatives introduced in Section 4.9 also applies to coupled systems and can be used to derive the coupled direct and adjoint methods presented above. Furthermore, the UDE (4.70)

can also handle residual or functional components, as long as they are ultimately expressed as residuals.

> **Practical Tip 4.8.** 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[a] 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.
>
> [a] http://openmdao.org

4.11 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 require 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 scaling advantages. The disadvantage is that the method is strongly intrusive and often considerable development effort is required.

4.12 Further Notes

- Complex step has been generalized to work with complex functions (multicomplex step) [13] and to provide exact second derivatives [14].
- An approach related to the complex step and AD uses *dual numbers*. These are numbers of the form $x + \epsilon x'$, where $\epsilon^2 = 0$.
- Many AD tools can be applied recursively to yield higher order derivatives, although such an approach is not typically efficient and is sometimes unstable [15].
- Curtis et al. [16] were the first to show that the number of function evaluations could be reduced in evaluating the Jacobian for sparse matrices. Gebremedhin et al. [17] provide a review paper of graph coloring in the context of computing derivatives. Gray et al. [12] show how to use graph coloring to compute total coupled derivatives.
- Reverse mode AD is now widely used in the backpropagation step in machine learning, where the derivative of a scalar error function with respect to a large number of neural network weights is required [18].

Bibliography

[1] J. N. Lyness. Numerical algorithms based on the theory of complex variable. In *Proceedings — ACM National Meeting*, pages 125–133, Washington DC, 1967. Thompson Book Co.

[2] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967. ISSN 0036-1429 (print), 1095-7170 (electronic).

[3] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262, 2003. doi:10.1145/838250.838251. September.

[4] Andreas Griewank. *Evaluating Derivatives*. SIAM, Philadelphia, 2000.

[5] Uwe Naumann. *The Art of Differentiating Computer Programs—An Introduction to Algorithmic Differentiation*. SIAM, 2011.

[6] L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[7] Alexander B Wiltschko, Bart van Merriënboer, and Dan Moldovan. Tangent: automatic differentiation using source code transformation in Python. 2017.

[8] J. Revels, M. Lubin, and T. Papamarkou. Forward-Mode Automatic Differentiation in Julia. arXiv:1607.07892, July 2016.

[9] Richard D. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, jan 2010. doi:10.1137/080743627.

[10] Joaquim R. R. A. Martins and John T. Hwang. Review and unification of methods for computing derivatives of multidisciplinary computational models. *AIAA Journal*, 51(11):2582–2599, November 2013. doi:10.2514/1.J052184.

[11] John T. Hwang and Joaquim R. R. A. Martins. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM Transactions on Mathematical Software*, 44(4):Article 37, June 2018. doi:10.1145/3182393.

[12] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019. doi:10.1007/s00158-019-02211-z.

[13] Gregory Lantoine, Ryan P. Russell, and Thierry Dargent. Using multicomplex variables for automatic computation of high-order derivatives. *ACM Transactions on Mathematical Software*, 38(3):1–21, Apr 2012. ISSN 0098-3500. doi:10.1145/2168773.2168774.

[14] Jeffrey Fike and Juan Alonso. The development of hyper-dual numbers for exact second-derivative calculations. *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, Jan 2011. doi:10.2514/6.2011-886.

[15] Michael Betancourt. A geometric theory of higher-order automatic differentiation. arXiv:1812.11592 [stat.CO], Dec 2018.

[16] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse jacobian matrices. *IMA Journal of Applied Mathematics*, 13(1):117–119, Feb 1974. ISSN 1464-3634. doi:10.1093/imamat/13.1.117.

[17] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, Jan 2005. ISSN 1095-7200. doi:10.1137/s0036144504444711. URL http://dx.doi.org/10.1137/s0036144504444711.

[18] Atilim Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(1):5595—5637, January 2018.

CHAPTER 5

Constrained Gradient-Based Optimization

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 3 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 important part of the line search in both constrained optimization approaches.

5.1 Constrained Problem Formulation

We can express a general constrained optimization problem as

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{with respect to} \quad & x_i \qquad\qquad i = 1, \ldots, n_x \\ \text{subject to} \quad & g_j(x) \leq 0 \quad j = 1, \ldots, n_g \\ & h_k(x) = 0 \quad k = 1, \ldots, n_h \\ & \underline{x}_i \leq x_i \leq \overline{x}_i \quad i = 1, \ldots, n_x \end{aligned} \tag{5.1}$$

where the $g_j(x)$ are the *inequality constraints*, $h_k(x)$ are the *equality constraints*, and $\underline{x}$ and $\overline{x}$ 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 ϵ is some small number. In practice, numerical precision and the implementations of many methods make it desirable to distinguish between equality and inequality constraints.

> **Practical Tip 5.1.** Don't 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 7.

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.

> **Practical Tip 5.2.** Don't specify bounds as nonlinear constraints.
>
> Bounds are a special category and the simplest form of a constraint:
>
> $$\underline{x}_i \leq x_i \leq \overline{x}_i$$
>
> 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:

$$[\nabla h]_{ij} = \begin{bmatrix} \nabla h_1^T \\ \vdots \\ \nabla h_{n_h}^T \end{bmatrix} = \frac{\partial h_i}{\partial x_j}, \tag{5.2}$$

so this Jacobian is a rectangular 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 3.2). We begin with equality constraints because the mathematics and intuition is 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

$$f(x + p) \approx f(x) + \nabla f(x)^T p. \tag{5.3}$$

If x^* is a minimum point then every point in a small neighborhood must have a greater value,

$$f(x^* + p) \geq f(x^*). \tag{5.4}$$

Given the Taylor series expansion (5.3), the only way that this inequality can be satisfied is if

$$\nabla f(x^*)^T p \geq 0. \tag{5.5}$$

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. 5.1. If the problem is unconstrained then p can be in any direction and the

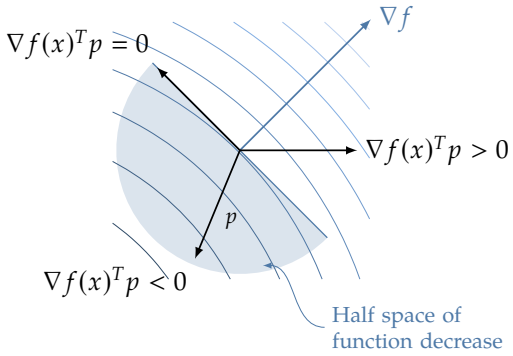


Figure 5.1: The gradient $f(x)$, which is the direction of steepest function increase, splits the design space into two halves. Here we highlight the halfspace of directions that result in function decrease.

only way to satisfy this inequality is if $\nabla f(x^*) = 0$. Therefore, $\nabla f(x^*) = 0$ is a necessary condition for an unconstrained minimum.

Now consider the constrained case. The function increase condition (5.5) 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

$$h_j(x + p) \approx h_j(x) + \nabla h_j(x)^T p, \qquad j = 1, \ldots, n_h. \tag{5.6}$$

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

$$\nabla h_j(x)^T p = 0, \qquad j = 1, \ldots, n_h, \tag{5.7}$$

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. 5.2 for 2-D and 3-D illustrations).

For a point to be a constrained minimum,

$$\nabla f(x^*)^T p \geq 0 \tag{5.8}$$

for all p such that

$$\nabla h_j(x^*)^T p = 0, \qquad j = 1, \ldots, n_h. \tag{5.9}$$

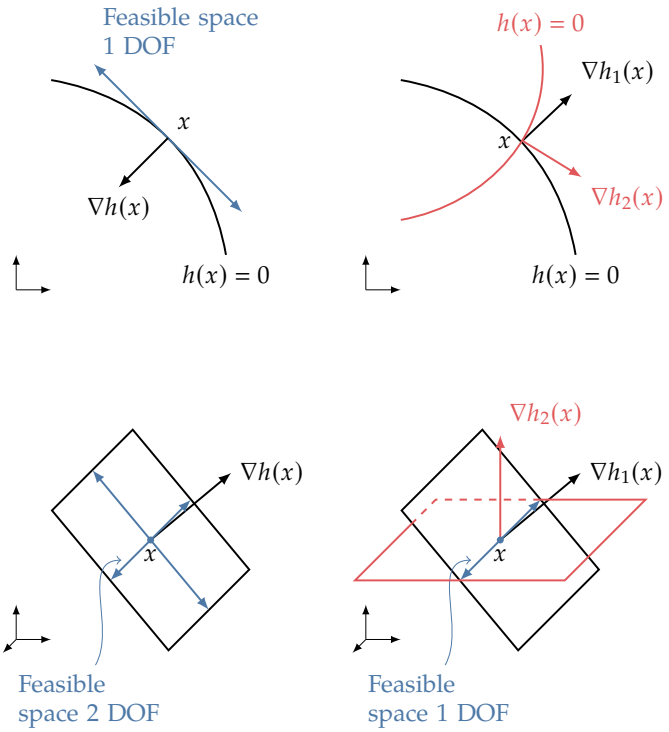


Figure 5.2: Feasible spaces for 2-D examples with one constraint (upper left), and two constraints (upper right); 3-D examples with two constraints (lower left) and one constraint (lower right).

As previously mentioned, the feasible directions form a hyperplane in n_x dimensions. The intersection of this hyperplane with the half 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 (5.8) 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. 5.3 for an illustration in 3-D space).

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.,

$$\nabla f(x^*) = -\sum_{j=1}^{n_h} \lambda_j \nabla h_j(x^*), \tag{5.10}$$

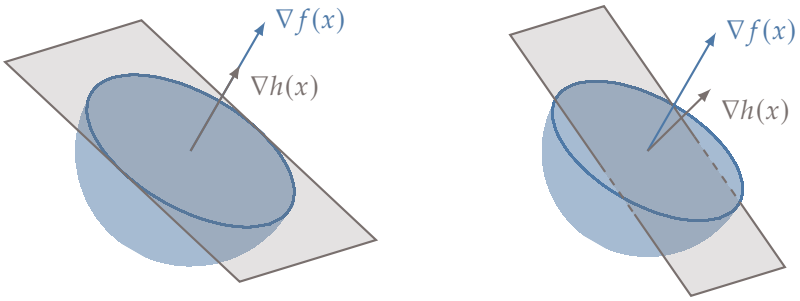


Figure 5.3: All feasible directions must be contained in the hyperplane perpendicular to the gradient of the objective function so that there are no feasible descent directions. Here we show 3-D example with one constraint for a point satisfying optimality (left) and a point not satisfying optimality (right).

where λ_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,

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T h(x), \tag{5.11}$$

where λ is the vector of Lagrange multipliers defined above, which are now unknown variables as well. 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 λ we need to set both partial derivatives equal to zero as follows,

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x_i} &= \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} = 0, \qquad && i = 1, \ldots, n_x, \\ \frac{\partial \mathcal{L}}{\partial \lambda_j} &= h_j = 0, && j = 1, \ldots, n_h. \end{aligned} \tag{5.12}$$

The first condition is the constrained optimality condition we explained previously in Eq. (5.17). The second enforces the equality constraints, which must 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, λ. 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 3 to solve the optimality

conditions (5.12), 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 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:

$$p^T \left[\nabla_{xx} \mathcal{L}\right] p > 0, \tag{5.13}$$

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

$$\nabla h_j(x)^T p = 0, \quad j = 1, \ldots, n_h. \tag{5.14}$$

That is, the feasible directions are in the null space of the Jacobian of the constraints.

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 (5.3), we get the condition

$$\nabla f(x^*)^T p \geq 0, \tag{5.15}$$

which is the same as for the equality constrained case.

To consider the constraints, we use the same linearization of the constraint (5.6), but now we enforce an inequality to get

$$g_j(x + p) \approx g_j(x) + \nabla g_j(x)^T p \leq 0, \qquad j = 1, \ldots, n_g. \tag{5.16}$$

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 only with a crucial modification. First, the requirement

that the gradient of the objective is a linear combination of the gradients of the constraints,

$$\nabla f(x^*) = -\sum_{j=1}^{n_g} \lambda_j \nabla g_j(x^*), \tag{5.17}$$

only needs to consider the active constraints. Second, the sign of the Lagrange multipliers is now significant. 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. 5.4.

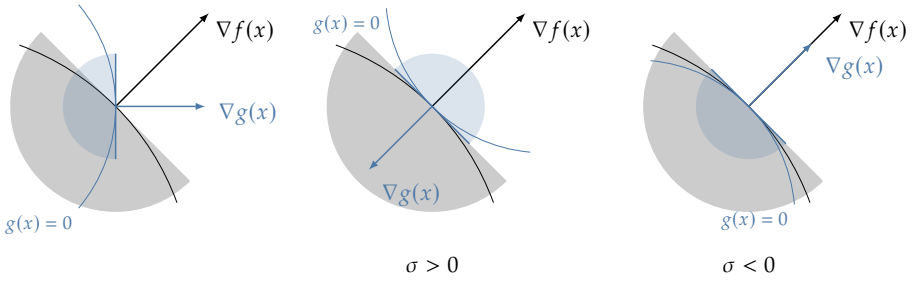


Figure 5.4: Constrained minimum conditions for 2-D case with one inequality constraint. The objective function gradient must be parallel and have opposite directions (corresponding to a positive Lagrange multiplier) so that there are no feasible descent directions.

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

$$g_k + s_k^2 = 0, \quad k = 1, \ldots, n_g \tag{5.18}$$

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 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

$$\mathcal{L}(x, \lambda, \sigma, s) = f(x) + \lambda^T h(x) + \sigma^T \left(g(x) + s^2 \right),$$

where σ 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

$$\nabla_x \mathcal{L} = 0 \Rightarrow \quad \frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} + \sum_{k=1}^{n_g} \sigma_k \frac{\partial g_k}{\partial x_i} = 0, \quad i = 1, \ldots, n_x \tag{5.19}$$

This criteria is the same as before, but with additional Lagrange multipliers and constraints. Taking the derivatives with respect to the equality Lagrange multipliers,

$$\nabla_\lambda \mathcal{L} = 0 \Rightarrow \quad \frac{\partial \mathcal{L}}{\partial \lambda_j} = h_j = 0, \quad j = 1, \ldots, n_h, \tag{5.20}$$

which enforced the equality constraints as before. Taking derivatives with respect to the inequality Lagrange multipliers, we get

$$\nabla_\sigma \mathcal{L} = 0 \Rightarrow \quad \frac{\partial \mathcal{L}}{\partial \sigma_k} = g_k + s_k^2 = 0 \quad k = 1, \ldots, n_g, \tag{5.21}$$

which enforces the inequality constraints. Finally, differentiating the Lagrangian with respect to the slack variables, we obtain

$$\nabla_s \mathcal{L} = 0 \Rightarrow \quad \frac{\partial \mathcal{L}}{\partial s_k} = 2\sigma_k s_k = 0, \quad k = 1, \ldots, n_g, \tag{5.22}$$

which is called the *complementarity condition*. This condition helps us to 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:

$$\begin{aligned} \frac{\partial f}{\partial x_i} + \sum_{j=1}^{n_h} \lambda_j \frac{\partial h_j}{\partial x_i} + \sum_{k=1}^{n_g} \sigma_k \frac{\partial g_k}{\partial x_i} &= 0, \quad i = 1, \ldots, n_x \\ h_j &= 0, \quad j = 1, \ldots, n_h \\ g_k + s_k^2 &= 0, \quad k = 1, \ldots, n_g \\ \sigma_k s_k &= 0, \quad k = 1, \ldots, n_g \\ \sigma_k &\geq 0, \quad k = 1, \ldots, n_g \end{aligned} \tag{5.23}$$

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. 5.4, 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. (5.13). 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,

$$\begin{aligned} \nabla h_j(x)^T p &= 0, \quad j = 1, \ldots, n_h, \\ \nabla g_i(x)^T p &= 0, \quad \text{for all } i \text{ in active set.} \end{aligned} \tag{5.24}$$

Practical Tip 5.3. 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 with the goal of meeting 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 wouldn't reject that design. Thus, we shouldn't formulate the constraint in a way that is unnecessarily restrictive.

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 assumed; an inactive constraint could be made active by changing its value by a large enough amount.

For an active constraint $g_i(x) \leq 0$, if its value is perturbed such that $g_i(x) \leq$

$\varepsilon||\nabla g_i(x^*)||$, then,

$$\frac{\mathrm{d}f\left(x^*(\varepsilon)\right)}{\mathrm{d}\varepsilon} = -\sigma_i^*||\nabla g_i(x^*)||. \tag{5.25}$$

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 quantifies 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, 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 6, 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

$$F(x) = f(x) + \mu P(x), \tag{5.26}$$

where $P(x)$ is a penalty function and the scalar μ is a penalty parameter. This is similar in form to the Lagrangian, but one difference is that a value for μ 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 μ 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 (see Fig. 5.5).

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 add terms that improve the numerical properties. Many other penalties are possible, such

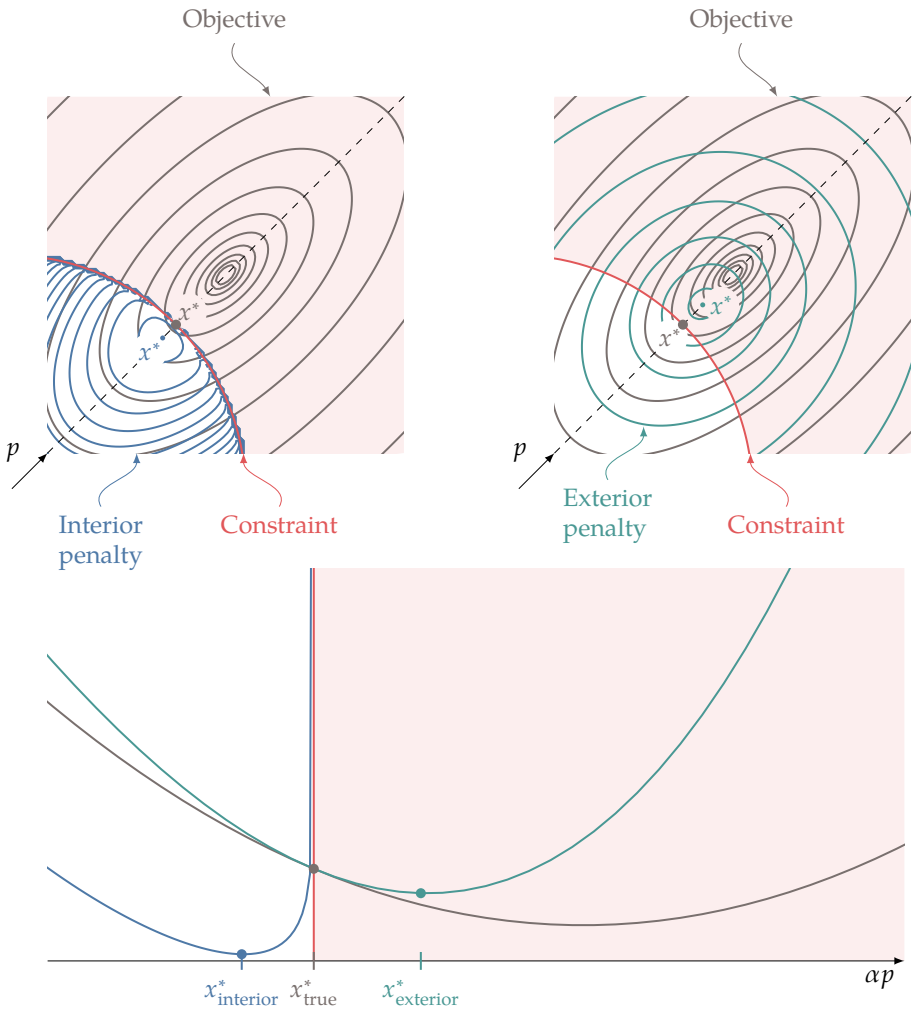


Figure 5.5: Interior penalties tend to infinity as the constraint is approached from the feasible side of the constraint (left), while exterior penalty functions activate when the points are not feasible (right). The minimum for both approaches is different from the true constrained minimum.

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,

$$F(x;\mu) = f(x) + \frac{\mu}{2}\sum_i h_i(x)^2. \tag{5.27}$$

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.

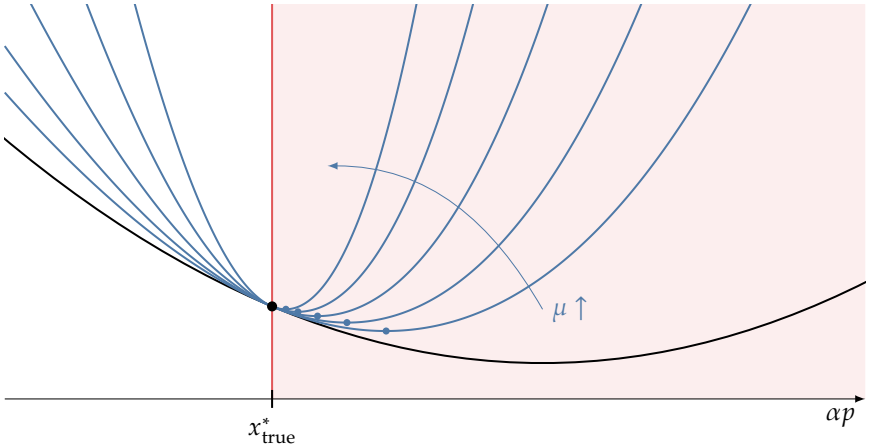


Figure 5.6: Quadratic penalty for an equality constrained 1-D problem. The minimum of the penalized function (blue dots) approaches the true constrained minimum (black circle) as the penalty parameter μ increases.

The value of the penalty parameter μ must be chosen carefully. Mathematically, we recover the exact solution to the constrained problem only as μ tends to infinity (see Fig. 5.6). However, starting with a large value for μ is not practical. This is because the larger the value of μ, 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 μ and solve the unconstrained problem. We then increase μ 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 Algorithm 14. By gradually increasing μ 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 14 Exterior penalty method.

Input: a starting point x_0
  while not converged **do**
    Minimize $F(x_k; \mu_k)$ with respect to x_k to yield x_k^*.
    Increase penalty parameter[1]: $\mu_{k+1} = \rho \mu_k$
    Update starting point for next optimization: $x_{k+1} = x_k^*$.
  end while
Return: "optimal" point x_{k+1}^* and corresponding function value $f(x_{k+1}^*)$

There are three potential issues with the approach outlined in Algorithm 14. If the starting value for μ 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 $\mu \to \infty$, 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 that for each constraint i,

$$h_i \approx \frac{\lambda_i^*}{\mu}. \tag{5.28}$$

Since $h_i = 0$ for the exact optimum, μ 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 μ. The added curvature is added in a direction perpendicular to the constraints, which makes the Hessian of the penalized function increasingly ill-conditioned as μ increases. When using Newton or quasi-Newton methods, such ill-conditioning causes numerical difficulties.

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 can be achieved by defining a new penalty function as

$$F(x; \mu) = f(x) + \frac{\mu}{2} \sum_j \max[0, g_j(x)]^2. \tag{5.29}$$

This 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.

[1] ρ may range from a conservative value (1.2) to an aggressive value (10), depending on the problem.

Practical Tip 5.4. Scaling is also important for constrained problems.

The same considerations on scaling discussed in Chapter 3 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 μ for constraint satisfaction, but the large μ degrades the numerical conditioning. The augmented Lagrangian method alleviates this dilemma by adding the quadratic penalty to the Lagrangian instead of the just adding it to the function. The augmented Lagrangian function for equality constraints is:

$$F(x, \lambda; \mu) = f(x) + \sum_j^{n_h} \lambda_j h_j(x) + \frac{\mu}{2} \sum_j^{n_h} h_j(x)^2, \tag{5.30}$$

Inequality constraints can be included in a similar way using the max function shown in Eq. (5.29). 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,

$$\nabla_x F(x, \lambda; \mu) = \nabla f(x) + \sum_j^{n_h} [\lambda_j + \mu h_j(x)] \nabla h_j = 0 \tag{5.31}$$

to those of the actual Lagrangian,

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = \nabla f(x^*) + \sum_j^{n_h} \lambda_j^* \nabla h_j(x^*) = 0, \tag{5.32}$$

which suggests the approximation

$$\lambda_j^* \approx \lambda_j + \mu h_j. \tag{5.33}$$

Therefore, we update the vector of Lagrange multipliers based on the current estimate of the Lagrange multipliers and constraint values using

$$\lambda_{k+1} = \lambda_k + \mu_k h(x_k) \tag{5.34}$$

The complete algorithm is shown in Algorithm 15.

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 μ too

Algorithm 15 Augmented Lagrangian penalty method.

Input: a starting point x_0, $\lambda_0 = 0$
  while not converged **do**
    Minimize $F(x_k, \lambda_k; \mu^k)$ with respect to x^k to yield x^{k^*}
    Update Lagrange multipliers: $\lambda_{k+1} = \lambda_k + \mu_k h(x_k)$
    Increase penalty parameter: $\mu_{k+1} = \rho \mu_k$
    Update starting point for next optimization: $x_{k+1} = x_k^*$
  end while
Return: optimal point x_{k+1}^* and corresponding function value $f(x_{k+1}^*)$

much. We can see this by comparing the augmented Lagrangian approximation to the constraints obtained from Eq. (5.33),

$$h_i \approx \frac{1}{\mu}(\lambda_i^* - \lambda_i) \tag{5.35}$$

with the corresponding approximation in the quadratic penalty method,

$$h_i \approx \frac{\lambda_i^*}{\mu}. \tag{5.36}$$

To drive the constraints to zero, the quadratic penalty relies solely on increasing μ 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 μ. Thus, the augmented Lagrangian can provide a good solution for x^* while avoiding the ill-conditioning issues of the quadratic penalty.

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,

$$P(x) = \sum_j^{n_g} -\frac{1}{g_j(x)}, \tag{5.37}$$

where $P(x) \to \infty$ as $c_i(x) \to 0^-$.

A more popular interior penalty function is the logarithmic barrier,

$$P(x) = \sum_{j}^{n_g} -\log\left(-g_j(x)\right), \tag{5.38}$$

which also approaches infinity as the constraint tends to zero from the feasible side. The penalty function is then,

$$F(x;\mu) = f(x) - \mu \sum_{j}^{n_g} \log(-g_j(x)). \tag{5.39}$$

Like exterior methods, interior methods must also solve a sequence of unconstrained problems but with $\mu \to 0$ (see Algorithm 16). 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. 5.7.

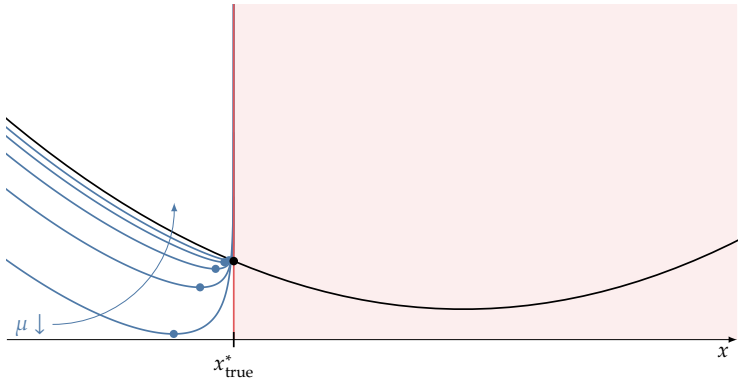


Figure 5.7: Logarithmic barrier penalty for an inequality constrained 1-D problem. The minimum of the penalized function (blue circles) approaches the true constrained minimum (black circle) as the penalty parameter μ decreases.

Algorithm 16 Interior penalty method.

Input: a starting point x_0
  while not converged **do**
    Minimize $F(x_k;\mu_k)$ with respect to x_k to yield x^*_k.
    Decrease penalty parameter[2]: $\mu_{k+1} = \rho\mu_k$
    Update starting point for next optimization: $x_{k+1} = x^*_k$.
  end while
Return: "optimal" point x^*_{k+1} and corresponding function value $f(x^*_{k+1})$

Example 5.1. Logarithmic interior penalty constraints.

As an example, consider the simple 2D quadratic function with one linear inequality constraint described below.

$$\begin{aligned} &\text{minimize} && (x_1 + 2)^2 + 3x_2^2 + 2x_1x_2 \\ &\text{subject to} && -2x_1 - x_2 \leq 0 \end{aligned} \tag{5.40}$$

Contours of the function and the constraint are shown in Fig. 5.8. Everything to the right of the line is feasible. The starred location is the constrained optimum. This problem is simple enough that we can solve it analytically: $x^* = (-2/9, 4/9)$.

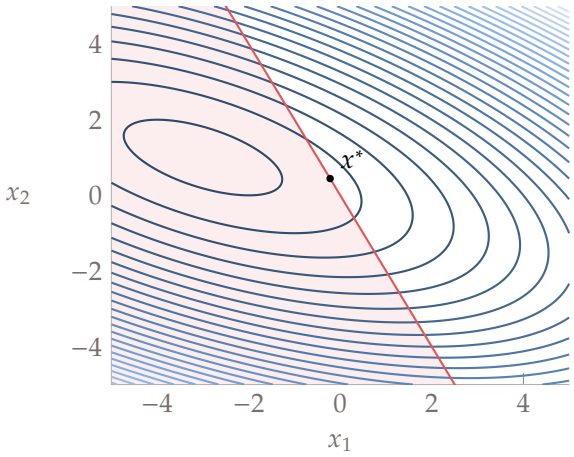


Figure 5.8: An example quadratic objective with linear constraint. Everything to the right of the constraint line is feasible and the constrained minimum is at the starred location.

Now let's form an interior penalty function using the log form. We start with a penalty parameter of $\mu = 1.0$. Level curves for this function are shown on the left side of Fig. 5.9. The minimum is clearly visible, though it is far from the true optimum. The scale on this plot is zoomed in relative to the previous plot so that an excessive number of contours aren't required to see the minimum. Note that as the constraint boundary is approached the level curves are tightly bunched indicating rapid change. To see this more clearly, the left side of Fig. 5.9 contains the s line segment, while the right figure plot the function value along this line segment. As the constraint is approached a "barrier" prevents the function from crossing the constraint.

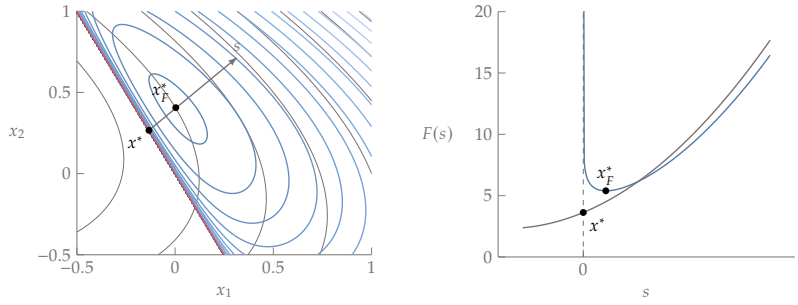


Figure 5.9: Log interior penalty for the example function with $\mu = 1$.

We repeat this process in Fig. 5.10 with a decreased penalty parameter of $\mu = 0.1$. Notice that the optimum of the penalty function is closer to the true constrained optimum, and that the barrier is acting over a smaller region, but is becoming increasingly sharp (the function is again zoomed in over a smaller area). Repeating this process will allow for convergence to the constrained optimum, although like the exterior penalty methods we generally need to settle for less strict convergence tolerances.

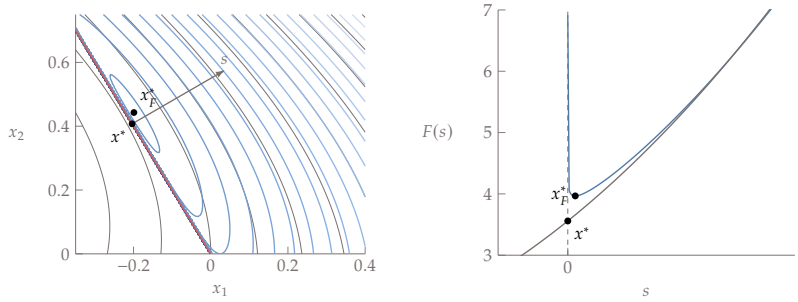


Figure 5.10: Log interior penalty for the example function with $\mu = 0.1$.

The methodology is essentially the same as is described in Algorithm 14, but with a decreasing penalty parameter. One major weakness of the method is that the penalty function is not defined for infeasible points and 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 or equal than zero.

Another weakness is that similarly to the exterior penalty methods, the Hessian becomes ill-conditioned as the penalty parameter tends to zero[1].

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) [2]. However, these methods have been superseded by modern interior point methods discussed in Section 5.5 and so we do not elaborate on further improvements to classical interior barrier methods.

Both interior and exterior penalties are demonstrated on a two-dimensional function in Fig. 5.5. Notice again that the exterior penalty leads to solutions that are slightly infeasible, and an interior penalty leads to a feasible solution but underpredicts the objective.

Practical Tip 5.5. Aggregate constraints.

As discussed in Chapter 4, adjoint methods, or reverse mode AD, are effective for scenarios with many inputs and few outputs. These methods are desirable as 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 [3], which acts like a soft-max:

$$KS(x) = \frac{1}{\rho} \ln \left(\sum_{j=1}^{m} e^{\rho g_j(x)} \right) \tag{5.41}$$

where ρ is an aggregation parameter, like a penalty parameter used in penalty methods.

5.4 Sequential Quadratic Programming

Sequential quadratic programming (SQP) is the first of the modern constrained optimization methods we discuss. 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:

$$\mathcal{L}(x, \lambda) = f(x) + h^T \lambda \tag{5.42}$$

The KKT conditions are that the derivatives of this function are zero:

$$\mathcal{R} = \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda) \\ \nabla_\lambda \mathcal{L}(x, \lambda) \end{bmatrix} = \begin{bmatrix} \nabla f(x) + [\nabla h(x)]^T \lambda \\ h(x) \end{bmatrix} = 0 \tag{5.43}$$

Recall that to solve a system of equations $R(u) = 0$ using Newton's method, we solve the sequence of linear systems

$$[\nabla_u \mathcal{R}(u_k)]\, s_u = -\mathcal{R}(u_k), \tag{5.44}$$

where $s_u = u_{k+1} - u_k$, and in our case $u = [x^T, \lambda^T]^T$. Differentiating the residual vector Eq. (5.43) with respect to the two concatenated vectors in u yields the block linear system,

$$\begin{bmatrix} \nabla^2_{xx} \mathcal{L} & [\nabla h]^T \\ [\nabla h] & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix} \tag{5.45}$$

This is a linear system of $n_x + n_h$ equations where the Jacobian matrix is square. 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:

$$x_{k+1} = x_k + s_x \tag{5.46}$$

$$\lambda_{k+1} = \lambda_k + s_\lambda \tag{5.47}$$

SQP can be derived in an alternative way that leads to different insights. This alternate approach requires an understanding of quadratic programming (QP). 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:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} x^T Q x + d^T x \\ \text{subject to} \quad & Ax + b = 0 \end{aligned} \tag{5.48}$$

Notice that the objective is quadratic in x and the constraint is linear in x. The constraint is a matrix equation and so 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:

$$\mathcal{L}(x, \lambda) = \frac{1}{2} x^T Q x + d^T x + \lambda^T (Ax + b) \tag{5.49}$$

We now take the partial derivatives and set them equal to zero:

$$\begin{aligned} \nabla_x \mathcal{L} &= Qx + d + A^T \lambda = 0 \\ \nabla_\lambda \mathcal{L} &= Ax + b = 0 \end{aligned} \tag{5.50}$$

We can express those same equations in a block matrix form:

$$\begin{bmatrix} Q & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -d \\ -b \end{bmatrix} \tag{5.51}$$

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[3]. 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 the following QP:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} s^T \nabla^2_{xx} \mathcal{L} s + \nabla_x \mathcal{L}^T s \\ \text{with respect to} \quad & s \\ \text{subject to} \quad & [\nabla h] s + h = 0 \end{aligned} \tag{5.52}$$

We substitute the gradient of the Lagrangian into the objective:

$$\frac{1}{2} s^T \nabla^2_{xx} \mathcal{L} s + \nabla f^T s + \lambda^T [\nabla h] s \tag{5.53}$$

Then, we substitute the constraint $[\nabla h] s = -h$ into the objective:

$$\frac{1}{2} s^T \nabla^2_{xx} \mathcal{L} s + \nabla f^T s - \lambda^T h \tag{5.54}$$

[3] In other words, this is a *convex* problem. Convex optimization is discussed in Chapter 9.

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:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} s^T \nabla^2_{xx} \mathcal{L} s + \nabla f^T s \\ \text{with respect to} \quad & s \\ \text{subject to} \quad & [\nabla h] s + h = 0 \end{aligned} \tag{5.55}$$

Using the QP solution method outlined above, results in the following system of linear equations:

$$\begin{bmatrix} \nabla^2_{xx} \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} -\nabla f \\ -h \end{bmatrix} \tag{5.56}$$

We replace $\lambda_{k+1} = \lambda_k + s_\lambda$ and multiply through:

$$\begin{bmatrix} \nabla^2_{xx} \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} + \begin{bmatrix} [\nabla h]^T \lambda_k \\ 0 \end{bmatrix} = \begin{bmatrix} -\nabla f \\ -h \end{bmatrix} \tag{5.57}$$

Subtracting the second term on both sides yields the same set of equations we found from applying Newton's method to the KKT conditions:

$$\begin{bmatrix} \nabla^2_{xx} \mathcal{L} & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix} \tag{5.58}$$

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 Algorithm 17 treating the active constraints as equality constraints and ignore 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 (5.45) 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 ϵ 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 α_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 α_{max}, which is the largest step size for which the constraints are still feasible. The line search then enforces α_{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 α_{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.

Practical Tip 5.6. How to handle max and min 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:

$$\max(\sigma) \leq \sigma_y \tag{5.59}$$

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:

$$\sigma_i \leq \sigma_y \text{ for } i = 1 \ldots m \tag{5.60}$$

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 (Tip 5.5) could be used.

5.4.3 Quasi-Newton SQP

The SQP method as discussed so far requires the Hessian of the Lagrangian $\nabla^2_{xx}\mathcal{L}$, 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 3, 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 Algorithm 17. 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.

Algorithm 17 SQP with quasi-Newton approximation.

Input: a starting point x_0, initial Lagrange multipliers λ_0, optimality tolerance τ_{opt}, feasibility tolerance τ_{feas}
$k = 0$
Evaluate f_0, ∇f_0, h_0, ∇h_0
while $||\nabla_x \mathcal{L}||_\infty > \tau_{\text{opt}}$ or $||h||_\infty > \tau_{\text{feas}}$ **do**
Solve the KKT system (5.45) for s_x and s_λ

$$\begin{bmatrix} W_k & [\nabla h]^T \\ \nabla h & 0 \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \mathcal{L} \\ -h \end{bmatrix}$$

Perform a line search in the direction of $p_k = s_x$, starting with $\alpha = 1$
Update step: $x_{k+1} = x_k + \alpha p_k$ and active set
Update the Lagrange multipliers: $\lambda_{k+1} = \lambda_k + s_\lambda$
Compute quasi-Newton approximation W_{k+1} using Eq. (5.61)
$k = k + 1$
end while
Return: optimal point x^* and corresponding function value $f(x^*)$

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):

$$W_{k+1} = W_k - \frac{W_k s_k s_k^T W_k}{s_k^T W_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{5.61}$$

where:

$$\begin{aligned} s_k &= x_{k+1} - x_k \\ y_k &= \nabla_x \mathcal{L}(x_{k+1}, \lambda_{k+1}) - \nabla_x \mathcal{L}(x_k, \lambda_{k+1}). \end{aligned} \tag{5.62}$$

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 [4]. This method replaces the y with a new vector r defined as:

$$r_k = \theta_k y_k + (1 - \theta_k) W_k s_k, \tag{5.63}$$

where the scalar θ_k is defined as

$$\theta_k = \begin{cases} 1 & \text{if} \quad s_k^T y_k \geq 0.2 s_k^T W_k s_k, \\ \frac{0.8 s_k^T W_k s_k}{s_k^T W_k s_k - s_k^T y_k} & \text{if} \quad s_k^T y_k < 0.2 s_k^T W_k s_k, \end{cases} \tag{5.64}$$

which can range from 0 to 1. We then use the same BFGS update formula Eq. (5.61) 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 θ. If $\theta_k = 0$ then Eq. (5.63) in combination with Eq. (5.61) 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 θ_k provides a linear weighting between keeping the current Hessian approximation and a full BFGS update.

The definition of θ_k (5.64) 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.

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:

$$\begin{aligned} \text{minimize} \quad & f(x) - \mu \sum_k \log s_k \\ \text{with respect to} \quad & x, s \\ \text{subject to} \quad & h_j(x) = 0 \text{ for } j = 1 \dots n_h \\ & g_k(x) + s_k = 0 \text{ for } k = 1 \dots \hat{n}_g \end{aligned} \tag{5.65}$$

One difference relative to the barrier method important distinction is that rather than treat 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 $\mu \to 0$. We will need to solve a sequence of solutions to this problem with μ approaching zero. Unlike SQP, this formulation uses only equality constraints and so it avoids the combinatorial problem of trying to determine an active set.

The KKT conditions for this problem are (forming the Lagrangian and taking derivatives with respect to x, λ, σ, s respectively):

$$\begin{aligned} \frac{\partial f}{\partial x_i} + \lambda_j \frac{\partial h_j}{\partial x_i} + \sigma_k \frac{\partial g_k}{\partial x_i} &= 0 \\ h_j &= 0 \\ g_k + s_k &= 0 \\ -\mu \sum_k \frac{1}{s_k} + \sigma_k &= 0 \end{aligned} \tag{5.66}$$

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 $\mathbf{1}$.

$$\begin{aligned} \nabla f(x) + [\nabla h(x)]^T \lambda + [\nabla g(x)]^T \sigma &= 0 \\ h &= 0 \\ g + s &= 0 \\ -\mu S^{-1} \mathbf{1} + \sigma &= 0 \end{aligned} \tag{5.67}$$

This form of the equation can pose numerically difficulties as $s \to 0$ (perhaps most obvious by examining the last equation in Eq. (5.66)), and so we will multiply the last equation through by S.

$$\begin{aligned} \nabla f(x) + [\nabla h(x)]^T \lambda + [\nabla g(x)]^T \sigma &= 0 \\ h &= 0 \\ g + s &= 0 \\ -\mu \mathbf{1} + S\sigma &= 0 \end{aligned} \tag{5.68}$$

We now have a set of residual equations and can apply Newton's method, just like we did for SQP. The result is:

$$\begin{bmatrix} \nabla_{xx}\mathcal{L}(x) & [\nabla h(x)]^T & [\nabla g(x)]^T & 0 \\ \nabla h(x) & 0 & 0 & 0 \\ \nabla g(x) & 0 & 0 & I \\ 0 & 0 & S & \Sigma \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \\ s_\sigma \\ s_s \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda, \sigma) \\ h(x) \\ g(x) + s \\ S\sigma - \mu \mathbf{1} \end{bmatrix} \tag{5.69}$$

where Σ is a diagonal matrix with σ 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:

$$\begin{bmatrix} \nabla_{xx}\mathcal{L}(x) & [\nabla h(x)]^T & [\nabla g(x)]^T & 0 \\ \nabla h(x) & 0 & 0 & 0 \\ \nabla g(x) & 0 & 0 & I \\ 0 & 0 & I & S^{-1}\Sigma \end{bmatrix} \begin{bmatrix} s_x \\ s_\lambda \\ s_\sigma \\ s_s \end{bmatrix} = - \begin{bmatrix} \nabla_x \mathcal{L}(x, \lambda, \sigma) \\ h(x) \\ g(x) + s \\ \sigma - \mu S^{-1} \mathbf{1} \end{bmatrix} \tag{5.70}$$

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.

Like quasi-Newton SQP, one generally uses an estimate of the Hessian of the Lagrangian and performs a line search. The basic procedure is like that of SQP (Algorithm 17) except that the penalty parameter needs to be decreased, and there is no active set to update.

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 [5]. 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.

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. 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:

$$F(x;\mu) = f(x) + \mu||c_{vio}(x)||_p \text{ for } p = 1 \text{ or } 2, \tag{5.71}$$

and the augmented Lagrangian:

$$F(x;\mu) = f(x) + \lambda(x)^T h(x) + \frac{1}{2}\mu||c_{vio}(x)||_2^2. \tag{5.72}$$

where c_{vio} are the constraint violations defined as:

$$c_{vio,i}(x) = \begin{cases} |h_i(x)| & \text{for equality constraints} \\ \max(0, g_i(x)) & \text{for inequality constraints} \end{cases} \tag{5.73}$$

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 [6]. Filter methods, in general, provide less unnecessary interference from taking a full Newton step [7], and have been shown to be quite effective for both sequential quadratic programming and interior point methods [8]. The approach is based on concepts from multiobjective optimization, which is discussed in more detail in Chapter 7. The basic idea is that we will 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, and any dominated points are removed from the filter (as they are no longer necessary).

Example 5.2. 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, let's 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. 5.11 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's 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.

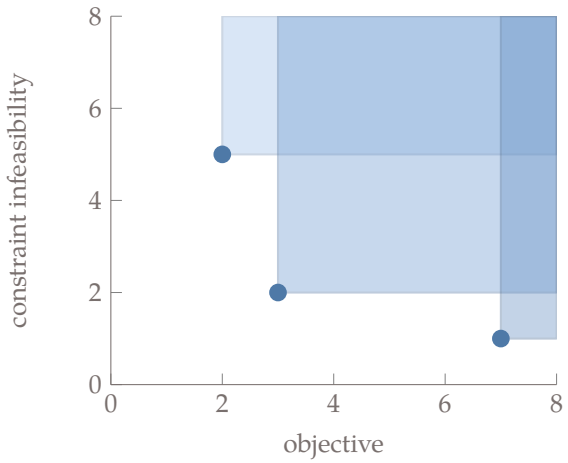


Figure 5.11: A filter method example showing three points in the filter, the shaded regions correspond to points that are dominated by the filter.

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 [7].

Practical Tip 5.7. 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 move

useful areas of the design space. Similarly, one should consider whether residuals should be solved internally or posed as constraints at the optimizer level.

5.7 Summary

With constraints we can no longer use ∇f as a first-order convergence criteria, 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.

5.8 Further Notes

- The damped BFGS formula it not always the best approach for all problems, and other Lagrangian Hessian approximation methods exist like the symmetric rank-one approximation [9]. Additionally, for very large problems storing a dense Hessian may be problematic and so limited-memory update methods are used [10].
- For the SQP methods, sometimes linearizing the constraints can lead to an infeasible QP subproblem, and additional techniques are needed to treat these subproblems [5, 11].
- The interior point methods are only outlined in this text. Many important implementation details and variations exist on things like: how often and in what manner to update the penalty parameter μ, 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 [12, 13].

Bibliography

[1] W. Murray. Analytical expressions for the eigenvalues and eigenvectors of the hessian matrices of barrier and penalty functions. *Journal of Optimization Theory and Applications*, 7(3):189–196, mar 1971. doi:10.1007/bf00932477.

[2] Anders Forsgren, Philip E. Gill, and Margaret H. Wright. Interior methods for nonlinear optimization. *SIAM Review*, 44(4):525–597, jan 2002. doi:10.1137/s0036144502414942.

[3] G. Kreisselmeier and R. Steinhauser. Systematic control design by optimizing a vector performance index. *IFAC Proceedings Volumes*, 12(7):113–117, Sep 1979. ISSN 1474-6670. doi:10.1016/s1474-6670(17)65584-8.

[4] M. J. D. Powell. Algorithms for nonlinear constraints that use lagrangian functions. *Mathematical Programming*, 14(1):224–248, dec 1978. doi:10.1007/bf01588967.

[5] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, 2nd edition, 2006.

[6] Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. *Mathematical Programming*, 91(2):239–269, jan 2002. doi:10.1007/s101070100244.

[7] Roger Fletcher, Sven Leyffer, and Philippe Toint. A brief history of filter methods. ANL/MCS-P1372-0906, Argonne National Laboratory, Sep 2006.

[8] Hande Y. Benson, Robert J. Vanderbei, and David F. Shanno. Interior-point methods for nonconvex nonlinear programming: Filter methods and merit functions. *Computational Optimization and Applications*, 23(2):257–272, 2002. doi:10.1023/a:1020533003783.

[9] Roger Fletcher. *Practical Methods of Optimization*. Wiley, 2nd edition, 1987.

[10] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, aug 1989. doi:10.1007/bf01589116.

[11] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1): 99–131, 2005. doi:10.1137/S0036144504446096.

[12] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, apr 2005. doi:10.1007/s10107-004-0559-y.

[13] Richard H. Byrd, Mary E. Hribar, and Jorge Nocedal. An interior point algorithm for large-scale nonlinear programming. *SIAM Journal on Optimization*, 9(4):877–900, jan 1999. doi:10.1137/s1052623497325107.

CHAPTER 6

Gradient-Free Optimization

Gradient-free algorithms fill an important role in optimization. The gradient-based algorithms introduced in Chapter 3 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 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.

6.1 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 4.4.2). 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 es-

sential when deriving the optimality conditions, both for unconstrained functions and constrained functions. More specifically, the KKT conditions (5.12) 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 KTT 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 3.2. 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.

> **Practical Tip 6.1.** 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.

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 minima. 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.

Another oft-cited reason for using a gradient-free method is because there are 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 7.

Another common reason for using gradient-free methods is because there are discrete 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 10). Some of the algorithms introduced in this chapter can handle discrete variables directly. A broader discussion of discrete optimization is contained in Chapter 10.

The proceeding discussion highlights that multimodality, multiple objectives, or discrete variables, though commonly mentioned, are not necessarily definitive reasons for choosing a gradient-free algorithm. One of the most salient features when choosing between a gradient-free and a gradient-based approach is the dimension of the problem. Figure 6.1 shows how many function evaluations are required to minimize the ND 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 [1]. The general takeaway is that for problems of small size (usually ≤ 30 variables [2]) 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.

6.2 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 6.1 and classify them according to the characteristics introduced in Fig. 1.7.

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–Meade algorithm (which we

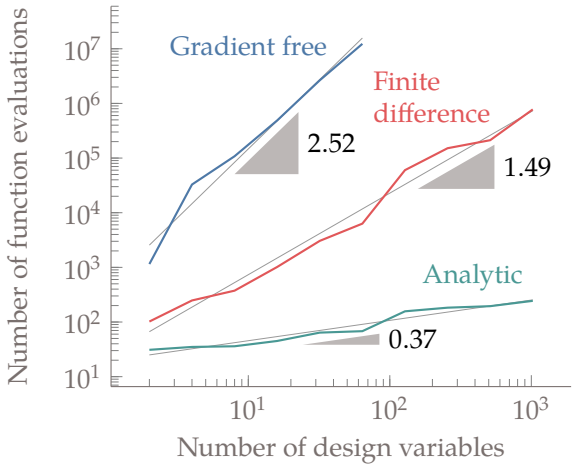


Figure 6.1: Cost of optimization for increasing number of design variables of the multidimensional Rosenbrock function. A gradient-free algorithm compared with and a gradient-based algorithm with gradients computed with finite-differences and analytically. A gradient-based optimizer with analytic gradients enables much better scalability.

detail in Section 6.3) 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. 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.

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 3.6 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 3.

Table 6.1: Classification of gradient-free optimization methods, using the characteristics of Fig. 1.7.

	Search		Optimal criteria		Iteration procedure		Function evaluation		Stochas -ticity	
	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		✓		✓		✓	✓			✓

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 6.6 and branch and bound search (discussed in Chapter 10)—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. Branch-and-bound search also partitions the design space, but estimates lower and upper bounds for the optimum by using the 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 12). 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. 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 are global. The stochastic elements in these algorithms contribute to a 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 because heuristic algorithms have a poor convergence rate, especially in higher dimensions, and because they lack a first-order optimality criterion.

Most evolutionary algorithms are inspired by natural processes, such as evolution, animal behavior, and physics. Nature-inspired algorithms have been a fertile area of developments, as a large number of algorithms have been proposed, fueled by a never-ending supply of biological phenomena for inspiration. These algorithms are more of an analogy of what happens in nature rather than an actual model because they are oversimplified models at best and incorrect at worst. 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".

In this chapter, we cover four gradient-free algorithms: the Nelder–Mead algorithm, a genetic algorithm, particle swarm optimization, and the DIRECT method.

6.3 Nelder–Mead Algorithm

The simplex method of Nelder and Mead [3] is a deterministic direct-search methods 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, $X = \left\{x^{(0)}, x^{(1)}, \ldots, x^{(n)}\right\}$. 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 four 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 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, $\left\{x^{(1)}, \ldots, x^{(n)}\right\}$, can be computed by

$$x^{(i)} = x^{(0)} + s^{(i)}, \tag{6.1}$$

where $s^{(i)}$ is a vector whose components j are defined by

$$s_j^{(i)} = \begin{cases} \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right), & \text{if } j = i \\ \frac{l}{n\sqrt{2}}\left(\sqrt{n+1}-1\right) + \frac{l}{\sqrt{2}}, & \text{if } j \neq i \end{cases} \tag{6.2}$$

For example, Fig. 6.2 shows a starting simplex for a two-dimensional problem.

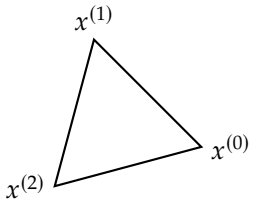


Figure 6.2: Starting simplex for $n = 2$.

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 $X = \left\{x^{(0)}, x^{(1)}, \ldots, x^{(n-1)}, x^{(n)}\right\}$, 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 show in Fig. 6.3. Each of these operations, except for the shrinking, generates a new point given by:

$$x = x_c + \alpha \left(x_c - x^{(n)} \right) \tag{6.3}$$

where α is a scalar, and x_c is the centroid of all the points except for the worst one, i.e.,

$$x_c = \frac{1}{n} \sum_{i=0}^{n-1} x^{(i)}. \tag{6.4}$$

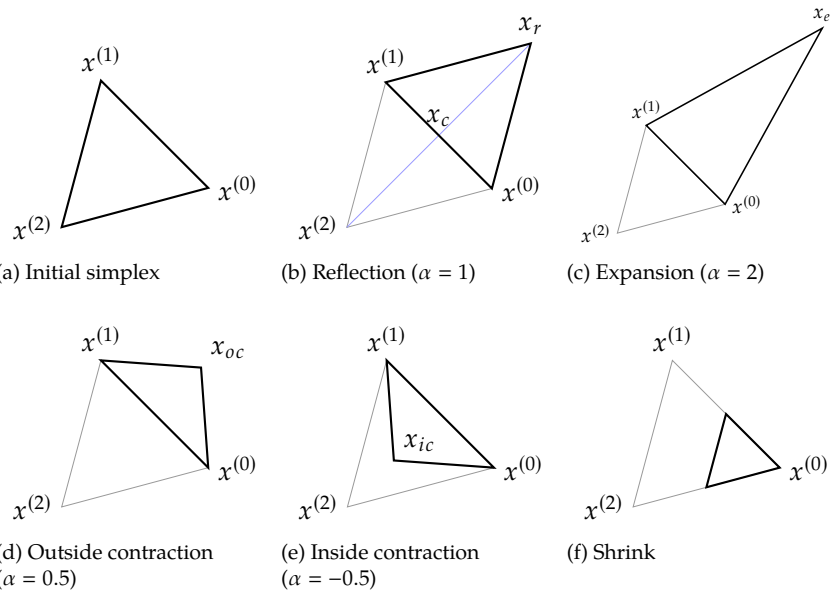
This generates a new point along the line that connects the worst point, $x^{(n)}$, and the centroid of the remaining points, x_c. This direction can be seen as a possible descent direction.

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. (6.3) with $\alpha = 1$ as shown in Fig. 6.3. 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. (6.3) 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,

$$x^{(i)} = x^{(0)} + \gamma \left(x^{(i)} - x^{(0)} \right) \quad \text{for } i = 1, \ldots, n, \tag{6.5}$$

where $\gamma = 0.5$.

Algorithm 18 details how a new simplex is obtained for each iteration. In each iteration the focus is to replacing the worst point with a better one, as opposed to improving the best. The corresponding flowchart is shown in Fig. 6.4.



(a) Initial simplex

(b) Reflection ($\alpha = 1$)

(c) Expansion ($\alpha = 2$)

(d) Outside contraction ($\alpha = 0.5$)

(e) Inside contraction ($\alpha = -0.5$)

(f) Shrink

Figure 6.3: Nelder–Mead algorithm operations for $n = 2$.

Algorithm 18 Nelder–Mead algorithm

Input: Initial guess, x_0
$k = 0$
Create a simplex with edge length l according to Eq. (6.1)

while simplex size (6.6) or standard deviation (6.7) are above tolerance **do**
Order the points from the lowest (best) function value to the highest (worst), $X = \left\{x^{(0)}, \ldots, x^{(n-1)}, x^{(n)}\right\}$.
Evaluate x_c, the centroid of the simplex point excluding the worst point $x^{(n)}$ (Eq. (6.4))
Perform *reflection* to obtain x_r, and evaluate $f(x_r)$

if $f(x_r) < f(x^{(0)})$ **then** ▷ Is reflected point is better than the best?
Perform *expansion* to obtain x_e
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
$x^{(n)} = x_r$ ▷ Accept reflection
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?
Perform an *inside contraction* to obtain x_{ic}
if $f(x_{ic}) < f(x^{(n)})$ **then** ▷ Inside contraction better than worst?
$x^{(n)} = x_{ic}$ ▷ Accept inside contraction
else
Shrink the simplex
end if
else
Perform an *outside contraction* to obtain x_{oc}
if $f(x_{oc}) < f(x_r)$ **then** ▷ O. contraction better than reflected?
$x^{(n)} = x_{oc}$ ▷ Accept outside contraction
else
Shrink the simplex
end if
end if
end if
$k = k + 1$
end while
Return: Optimum, x^*

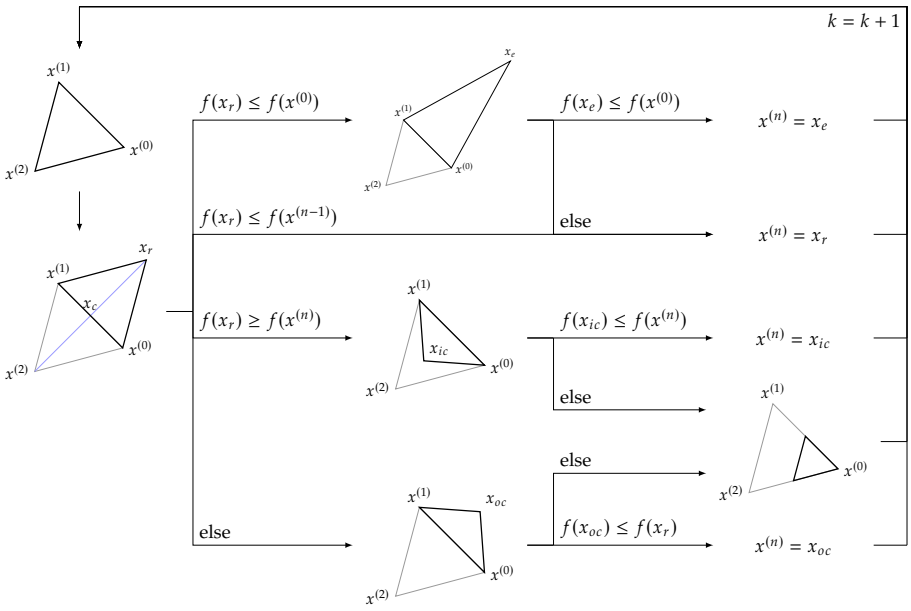


Figure 6.4: Flowchart of Nelder–Mead (Algorithm 18).

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.,

$$\Delta_x = \sum_{i=0}^{n-1} ||x^{(i)} - x^{(n)}||, \tag{6.6}$$

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,

$$\Delta_f = \sqrt{\frac{\sum_{i=0}^{n} \left(f^{(i)} - \bar{f}\right)^2}{n+1}}, \tag{6.7}$$

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 not be differentiable, so a linear penalty method would suffice.

Example 6.1. Nelder–Mead algorithm applied to the mild Rosenbrock function.

Figure 6.5 shows the sequence of simplices that results when minimizing the mil Rosenbrock 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 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_{best} and f_{worst} the problem took 62 function evaluations.

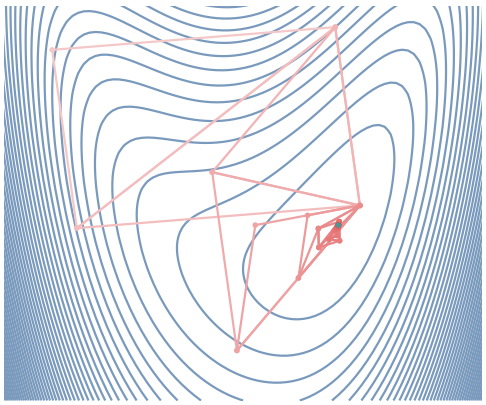


Figure 6.5: Sequence of simplices that minimize the mild Rosenbrock function

6.4 Genetic Algorithms

Genetic algorithms (GAs) are the most well-know and widely used type of evolutionary algorithm. They were also among the earliest to have been developed. 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. 6.6. Each design variable is represented by a *gene*. As we will see later, there are different ways for genes to represent the design variables.

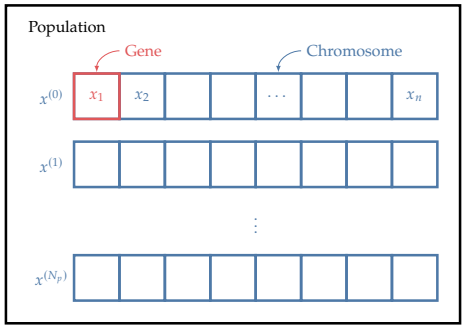


Figure 6.6: Each GA iteration involves a population of design points where each design is represented by a chromosome and each design variable is represented by a gene.

GAs evolve the population using an algorithm inspired by biological reproduction and evolution using three main steps: 1) selection, 2) 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 on 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.

Algorithm 19 and Fig. 6.7 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.

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 Algorithm 19 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

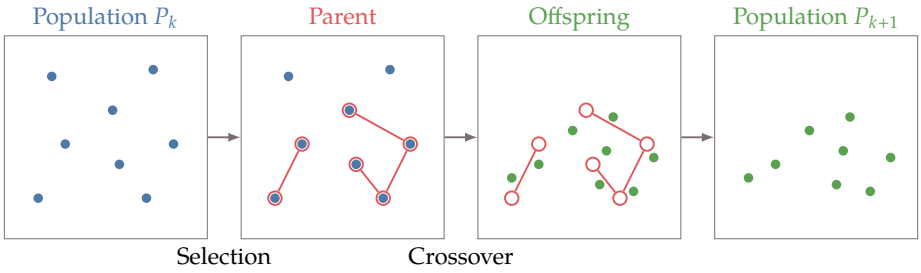


Figure 6.7: At each GA iteration, pairs of parents are selected from the population to generate the offspring through crossover, which become the new population.

Algorithm 19 Genetic algorithm

Input: Initial variables bounds
  $k = 0$
  Generate initial population $P_k = \left\{x^{(1)}, x^{(2)}, \ldots, x^{(N_P)}\right\}$
  while Stopping criterion is not satisfied **do**
    Compute objective $f(x)$ for each $x \in P_k$ ▷ Evaluate fitness
    Select $N_p/2$ parent pairs from P_k for crossover ▷ Selection
    Generate a new population of N_p offspring (P_{k+1}) ▷ Crossover
    Randomly mutate some points in the population ▷ Mutation
    $k = k + 1$
  end while
Return: "Optimum" (best result found), x^*

particular way of performing these operations for each of the possible design variable representations.

6.4.1 Binary-encoded Genetic Algorithms

The original genetic algorithms were based on binary encoding because they more naturally mimic chromosome encoding. Binary-coded GAs are still used, particularly for discrete or mixed-integer problems. 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

$$\Delta x = \frac{\bar{x} - \underline{x}}{2^m - 1}. \tag{6.8}$$

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

$$x = \underline{x} + \sum_{i=0}^{m-1} b_i 2^i \Delta x. \tag{6.9}$$

Example 6.2. 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. (6.8), 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. (6.9) 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 an 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 12.2).

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:

$$F = \frac{-f_i + \Delta F}{\max(1, \Delta F - f_{\text{low}})} \quad , \tag{6.10}$$

where $\Delta F = 1.1 f_{\text{high}} - 0.1 f_{\text{low}}$ 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 6.3.** Tournament selection process.
>
> Figure 6.8 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.

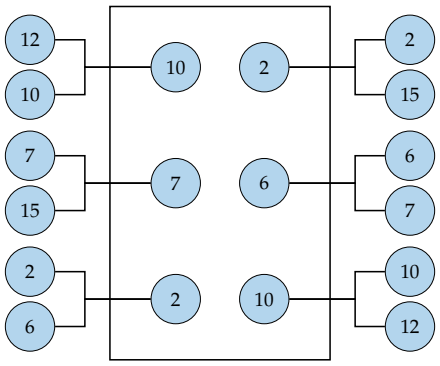


Figure 6.8: Tournament selection example.

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. (6.10). We then take the normalized cumulative sum of the scaled fitness values to compute an interval for each members in the population j as

$$S_j = \frac{\sum_{i=1}^{j} F_i}{\sum_{i=1}^{N_P} F_i} \tag{6.11}$$

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^{th} member is copied to the mating pool if

$$r \leq S_j \tag{6.12}$$

This ensures that the probability of a member being selected for reproduction is proportional to its scaled fitness value.

Example 6.4. 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. 6.9.

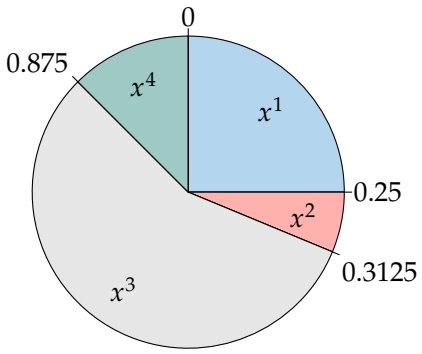


Figure 6.9: Roulette wheel selection example.

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 Fig. 6.10. 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.

Before crossover	After crossover
11 111	11 000
00 000	00 111

Figure 6.10: Single-point crossover operation example.

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 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 Fig. 6.11.

Before Mutation	After Mutation
11111	11011

Figure 6.11: Mutation example where only one bit changed.

6.4.2 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 continous, a real-encoded GA is generally more efficient than a binary-encoded GA [4]. We now describe the required changes to the GA operations in the real-encoded approach.

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,

$$x_i = \underline{x}_i + r(\bar{x}_i - \underline{x}_i) \tag{6.13}$$

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.

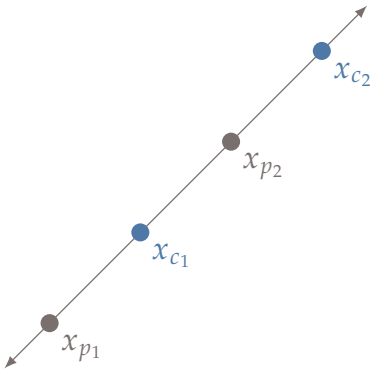


Figure 6.12: Linear crossover produces two new points along the line defined by the two parent points.

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 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:

$$\begin{aligned} x_{c_1} &= 0.5x_{p_1} + 0.5x_{p_2}, \\ x_{c_2} &= 2x_{p_2} - x_{p_1}, \end{aligned} \tag{6.14}$$

where parent two is more fit than parent 1 ($f(x_{p_2}) < f(x_{p_1})$). An example of this linear crossover approach is shown in Fig. 6.12, 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:

$$\begin{aligned} x_{p_1} &= [x_1, x_2, x_3, x_4] \\ x_{p_2} &= [x_5, x_6, x_7, x_8] \\ &\Downarrow \\ x_{c_1} &= [x_1, x_6, x_7, x_8] \\ x_{c_2} &= [x_5, x_2, x_3, x_4] \end{aligned} \tag{6.15}$$

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 [5, Ch. 4].

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:

$$x_{\text{new}\,i} = x_i + (r_i - 0.5)\Delta_i, \text{ for } i = 1 \ldots n_x \tag{6.16}$$

where r_i is a random number between 0 and 1, and Δ_i is a pre-selected maximum perturbation in the i^{th} direction. Many non-uniform methods exist as well. For example, we can use a Gaussian distribution

$$x_{\text{new}\,i} = x_i + \mathcal{N}(0, \sigma_i), \text{ for } i = 1 \ldots n_x \tag{6.17}$$

where σ_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 6.5. Genetic algorithm applied to Rosenbrock function.

Fig. 6.13 shows the evolution of the population when minimizing the Rosenbrock 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.

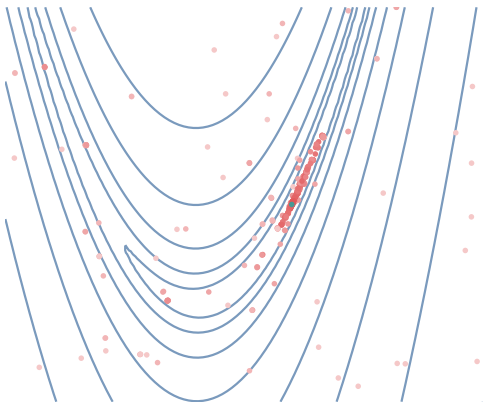


Figure 6.13: Population evolution using a genetic algorithm to minimize the Rosenbrock function

6.4.3 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.

6.4.4 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 criteria 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 can be a tricky criteria to use as, 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.

6.5 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.

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

$$x_{k+1}^{(i)} = x_k^{(i)} + v_{k+1}^{(i)} \Delta t, \tag{6.18}$$

where Δt is a constant artificial time step. The velocity for each particle is updated as follows:

$$v_{k+1}^{(i)} = \bar{w} v_k^{(i)} + c_1 r_1 \frac{x_{\text{best}}^{(i)} - x_k^{(i)}}{\Delta t} + c_2 r_2 \frac{x_{\text{best}} - x_k^{(i)}}{\Delta t}. \tag{6.19}$$

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 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_{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. (6.19) by Δt to yield a step

$$\Delta x_{k+1}^{(i)} = w \Delta x_k^{(i)} + c_1 r_1 \left(x_{\text{best}}^{(i)} - x_k^{(i)} \right) + c_2 r_2 \left(x_{\text{best}} - x_k^{(i)} \right). \tag{6.20}$$

We then use this step to update the particle position for the next iteration, i.e.,

$$x_{k+1}^{(i)} = x_k^{(i)} + \Delta x_{k+1}^{(i)}. \tag{6.21}$$

The three components of the update (6.20) are shown in Fig. 6.14 for a two-dimensional case.

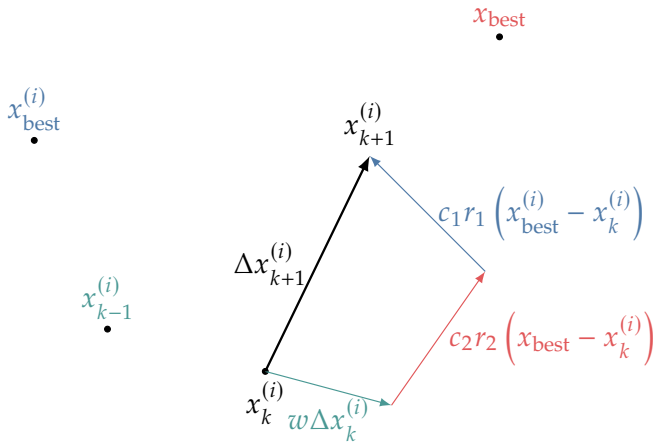


Figure 6.14: Components of the PSO update.

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 an 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 (Algorithm 20). 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 6.6. PSO algorithm applied to Rosenbrock function.

Figure 6.15 shows the sequence of simplices that results when minimizing the Rosenbrock 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.

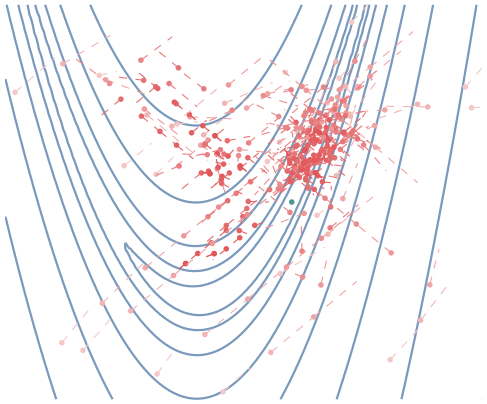


Figure 6.15: Sequence of particles that minimize the Rosenbrock function

6.6 DIRECT Algorithm

The divided rectangles (DIRECT) algorithm is different from the other gradient-free optimization algorithms in this chapter in that it is based on rigorous mathematics. 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*.

Algorithm 20 Particle swarm optimization algorithm

Input: Initial guess, x_0
Return: Optimum, x^*

$k = 0$
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 $f\left(x_0^{(i)}\right)$
  if i=0 **then**
    $x_{\text{best}} = x_0^{(i)}$
  else
    if $f\left(x_0^{(i)}\right) < f\left(x_{\text{best}}\right)$ **then**
      $x_{\text{best}} = x_0^{(i)}$
    end if
  end if
  Initialize "velocity" $\Delta x_k^{(i)}$
end for
repeat ▷ 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 $f\left(x_{k+1}^{(i)}\right)$
    if $f\left(x_{k+1}^{(i)}\right) < f\left(x_{\text{best}}^{(i)}\right)$ **then**
      $x_{\text{best}}^{(i)} = x_{k+1}^{(i)}$
    end if
  end for
  $k = k + 1$
until convergence criterion is met

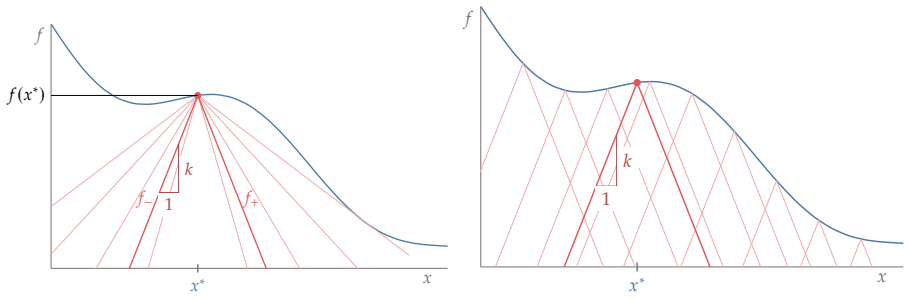


Figure 6.16: From a given trial point x^*, we can draw a cone with slope k (left). For a function to be Lipschitz continuous, we need all cones with slope k to lie under the function for all points in the domain (right).

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. 6.16. For a trial point x^*, we can draw a cone with slope k by drawing the lines,

$$f_+(x) = f(x^*) + k(x - x^*), \tag{6.22}$$
$$f_-(x) = f(x^*) - k(x - x^*), \tag{6.23}$$

to the left and right, respectively, We show this cone in Fig. 6.16 (left), as well as cones corresponding to other values of k.

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

$$|f(x) - f(x^*)| \leq k|x - x^*|, \quad \text{for all} \quad x, x^* \in D, \tag{6.24}$$

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. 6.16(right). Any k that satisfies condition (6.24) 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. 6.17. Using the property of the Lipschitz constant k defined in Eq. (6.24), we know that the function is always above a cone of slope k evaluated at any point in the domain.

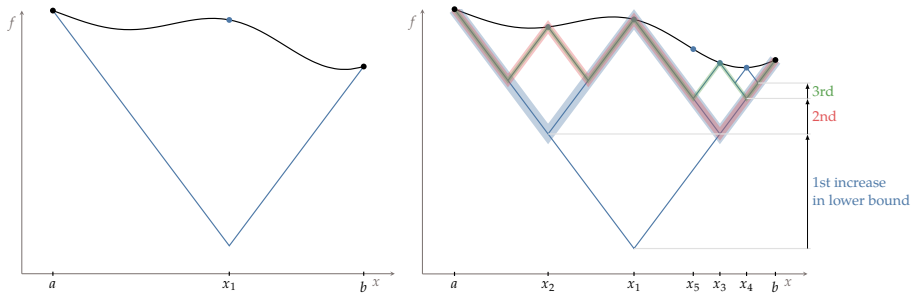


Figure 6.17: Shubert's algorithm requires an initial domain and a valid Lipschitz constant (left) and then increases the lower bound of the global minimum with each successive iteration (right).

We start by establishing a first lower bound on the global minimum by finding the intersection of the cones—x_1 in Fig. 6.17 (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 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. 6.18 (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 which we will not need, the lower bound on the minimum would be $f(c) - k(b-a)/2$.

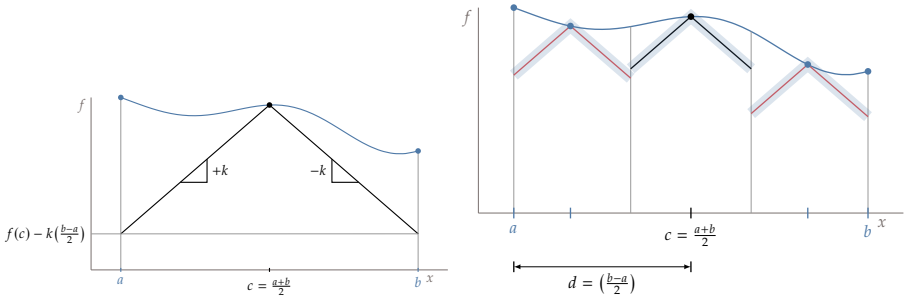


Figure 6.18: The DIRECT algorithm evaluates the middle point (left) and each successive iteration trisects the segments that have the greatest potential (right).

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. 6.18 (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. 6.18 (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. 6.19.

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

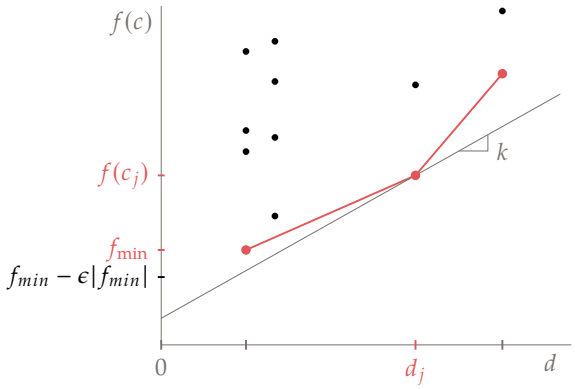


Figure 6.19: Potentially optimal segments in the DIRECT algorithm are identified by the lower convex hull of this plot.

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. 6.19, 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. 6.19. 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

$$f(c_j) - kd_j \leq f(c_i) - kd_i \quad \forall i \in S \tag{6.25}$$

$$f(c_j) - kd_j \leq f_{\min} - \varepsilon \left| f_{\min} \right| \tag{6.26}$$

where $f_{\min}$ is the best current objective function value, and ε is a small positive

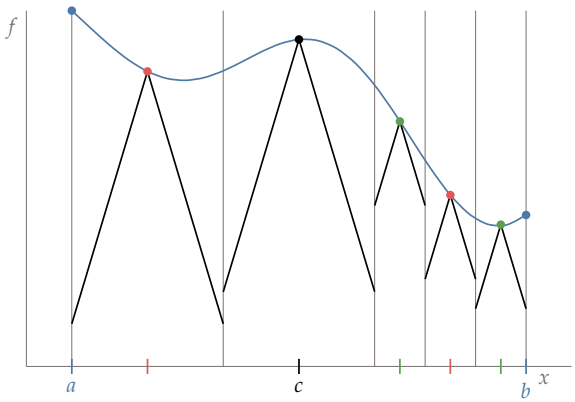


Figure 6.20: The lower bound on function values for the DIRECT method are discontinuous at the segment boundaries.

parameter. The first condition corresponds to finding the points in the lower convex hull mentioned previously.

The second condition in Eq. (6.26) 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 ε 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. 6.20.

DIRECT in n Dimensions

The n-dimensional DIRECT algorithm is similar to the one-dimensional version but becomes more complex. The main differences is that we deal with *hyperrectangles* instead of segments. A hyperrectagle can be defined by its centerpoint position c in n-dimensional space and a half length in each direction i, δe_i, as shown in Fig. 6.21. The DIRECT algorithm assumes that the initial dimensions are normalized so that we start with a hypercube.

To identify the *potentially optimal rectangles* at a given iteration, we use exactly the same conditions in Eqs. (6.25) and (6.26), but c_i is now the center of the

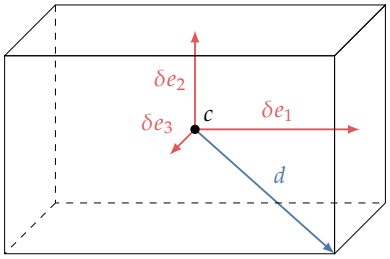


Figure 6.21: Hyperrectangle in three dimensions, where d is the maximum distance between the center and the vertices and δe_i is the half length in each direction i.

hyperrectangle, and d_i is the maximum distance from the center to a vertex. The explanation illustrated in Fig. 6.19 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. Algorithm 21 provides the details of this selection and its place in the overall algorithm.

Fig. 6.22 shows the first three iterations for a two-dimensional example and the corresponding visualization of conditions expressed in Eqs. (6.25) and (6.26). 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.

Algorithm 21 DIRECT in n-dimensions

Input: Bounds $\underline{x}$, $\overline{x}$
Return: Optimum, x^*

```
Normalize the design space to be the unit hypercube.
Compute center of the hypercube, c_0
f_min = f(c_0)
repeat
    Find the set S of potentially optimal hyperrectangles
    for each hyperrectangle r ∈ 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
until Converged
```

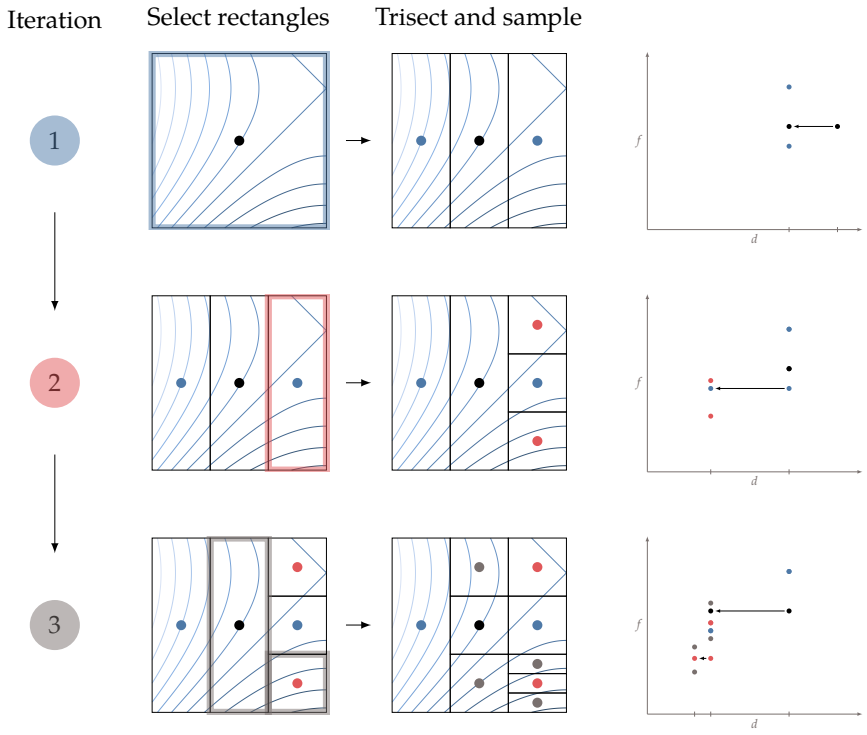
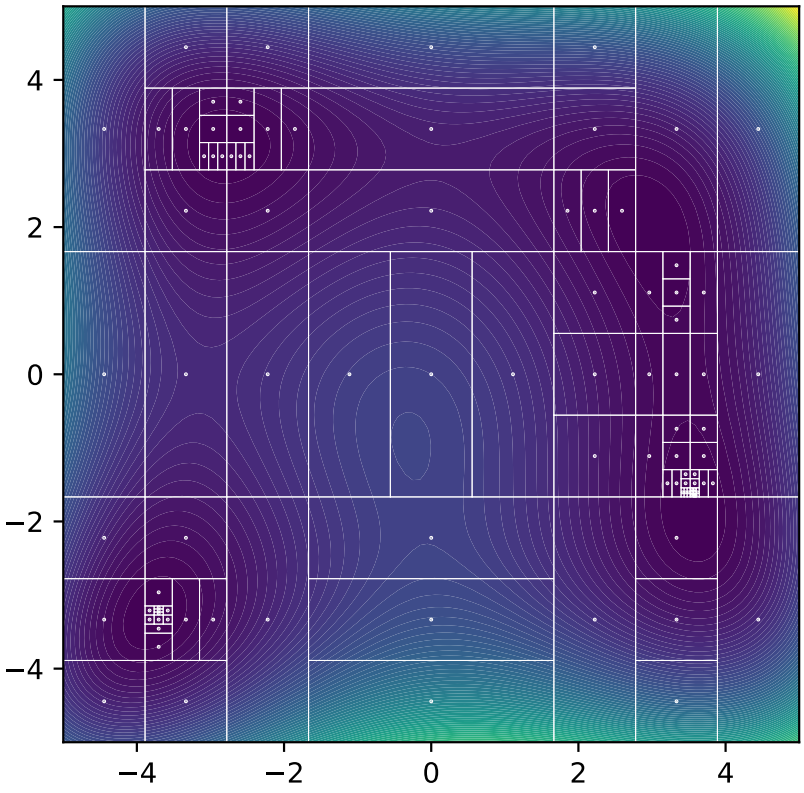


Figure 6.22: DIRECT iterations for two-dimensional case (left) and corresponding identification of potentially optimal rectangles (right).

Example 6.7. Minimization of Multimodal Function with DIRECT
Consider the two-dimensional analytic function

$$f(x_1, x_1) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2, \tag{6.27}$$

which has multiple local minima. Applying the DIRECT method to this function, we get the sequence of rectangles shown below.



6.7 Further Notes

- Rios and Sahinidis [2] review and benchmark a large selection of gradient-free optimization algorithms.
- The first GA software was written in 1954 [6, 7], followed by other seminal work. Initially, these GAs were not written to perform optimization, but rather, to model the evolutionary process. GAs were eventually applied to optimization [8]. Research in GAs increased dramatically in the two

decades after that, thanks in part to the exponential increase in computing power.

- Simon [4] covers the broad field of evolutionary algorithms, including the classic genetic algorithms and more recent algorithms such as PSO, ant colony optimization, and simulated annealing.
- NSGA-II is a popular genetic algorithm developed by Deb et al. [9], which can also handle multiobjective optimization, which we cover in Chapter 7.
- PSO was first proposed by [10]. Eberhart was an electrical engineer and Kennedy was a social-psychologist.
- There is a plethora of evolutionary algorithms in the literature, thanks to the fertile imagination of the research community. These 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, imperialst 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.
- The textbooks by Conn et al. [11] and Audet and Hare [12] provide a more extensive treatment of gradient-free optimization algorithms that are based on mathematical criteria.
- The DIRECT method was developed by Jones et al. [13], who was motivated to develop a global search that did not rely on any tunable parameters (such as population size in genetic algorithms). In this chapter, we presented an improved version [14]. 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, the details of which are described by Jones [14].
- NOMAD is an open-source gradient-free optimization software package [1]. It implements the mesh adaptive direct search algorithm (MADS), which is based on mathematical criteria [15].

[1] https://www.gerad.ca/nomad/

Bibliography

[1] Yin Yu, Zhoujie Lyu, Zelu Xu, and Joaquim R. R. A. Martins. On the influence of optimization algorithm and starting design on wing aerodynamic shape optimization. *Aerospace Science and Technology*, 75:183–199, April 2018. doi:10.1016/j.ast.2018.01.016.

[2] Luis Miguel Rios and Nikolaos V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56:1247–1293, 2013. doi:10.1007/s10898-012-9951-y.

[3] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[4] Dan Simon. *Evolutionary Optimization Algorithms*. John Wiley & Sons, Jun 2013. ISBN 1118659503.

[5] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Jul 2001. ISBN 047187339X.

[6] N. Barricelli. Esempi numerici di processi di evoluzione. *Methodos*, pages 45–68, 1954.

[7] D. B. Fogel. Nils barricelli—artificial life, coevolution, self-adaptation. *IEEE Computational Intelligence Magazine*, 1(1):41–45, Feb 2006. ISSN 1556-603X. doi:10.1109/MCI.2006.1597062.

[8] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, MI, 1975.

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002. doi:10.1109/4235.996017.

[10] R.C. Eberhart and J. A. Kennedy. New optimizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

[11] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, 2009.

[12] Charles Audet and Warren Hare. *Derivative-Free and Blackbox Optimization*. Springer, 2017. doi:10.1007/978-3-319-68913-5.

[13] D.R Jones, C.D. Perttunen, and B.E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Application*, 79(1):157–181, October 1993.

[14] Donald R. Jones. *Direct Global Optimization Algorithm*, pages 725–735. Springer US, Boston, MA, 2009. ISBN 978-0-387-74759-0. doi:10.1007/978-0-387-74759-0_128.

[15] S. Le Digabel. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software*, 37(4):1–15, 2011.

CHAPTER 7

Multiobjective Optimization

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.

7.1 Introduction

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. We can weight competing metrics, but combining the metrics in this way results in a single objective. We can also explore the tradeoffs between metrics, but once we make a design decision, we have indirectly created a single objective.

A common pitfall for beginning optimization practitioners is to categorize a problem as multiobjective without critical evaluation. When considering whether more than one objective should be used, one should ask whether or not there is a more fundamental underlying objective, or if some of the "objectives" are actually constraints.

Example 7.1. 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 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 and adding additional range doesn't really increase value (i.e., 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 potential objectives. 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 latter 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,

$$\text{minimize} \quad f(x) \tag{7.1}$$

becomes

$$\text{minimize} \quad F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}, \text{ where } n \geq 2 \tag{7.2}$$

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 of 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.

7.2 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 7.1 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 would 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. 7.2. The Pareto set refers to the vector of points x^*, whereas the Pareto front refers to the vector of functions $f(x^*)$.

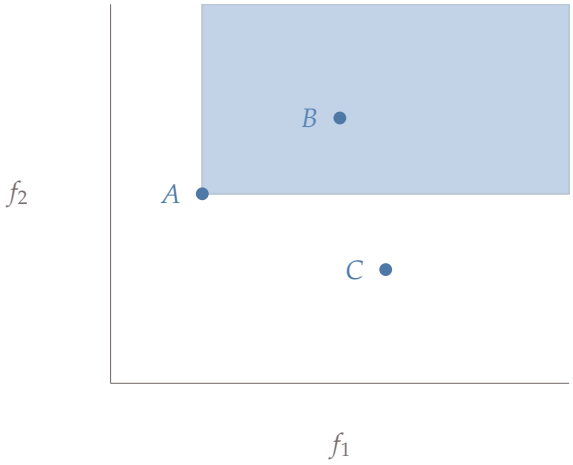


Figure 7.1: The three designs: A, B, and C, are plotted against two objectives: f_1 and f_2. The region in the shaded rectangle highlights points that are dominated by design A.

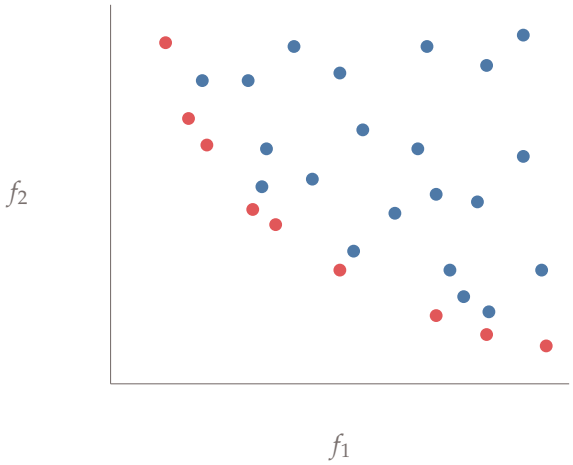


Figure 7.2: A plot of all the evaluated points in the design space plotted against two objectives f_1 and f_2. The set of red points are not dominated by any other and thus form the Pareto set.

Example 7.2. A Pareto front in wind farm optimization.

The Pareto front is a useful tool to produce design insights. Figure 7.3 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.

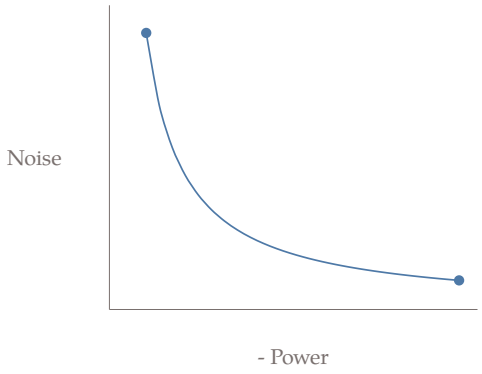


Figure 7.3: A notional Pareto front representing power and noise tradeoffs for a wind farm optimization problem.

7.3 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 constraint-epsilon method, the normal boundary interface method, and evolutionary algorithms.

7.3.1 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 frequently used. The idea is to combine all of the objectives into one objective using a weighted sum, which can be written as:

$$\bar{f}(x) = \sum_{i}^{N} w_i f_i(x), \tag{7.3}$$

where N is the number of objectives and the weights are usually normalized such that

$$\sum_{i}^{N} w_i = 1 \tag{7.4}$$

If we have two objectives, the objective reduces to

$$\bar{f}(x) = w f_1(x) + (1 - w) f_2(x), \tag{7.5}$$

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. 7.4). With the above form of the objective the slope of the line would be:

$$\frac{\mathrm{d}f_2}{\mathrm{d}f_1} = \frac{-w}{1 - w} \tag{7.6}$$

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. 7.5).

Using the Pareto front shown in Section 7.3.1, Fig. 7.5 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.

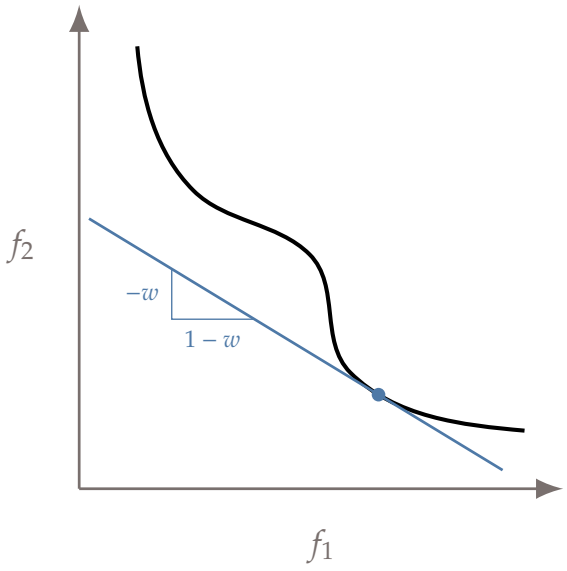


Figure 7.4: The black curve is the true Pareto front, and the blue curve represents the equation for the line defined by the weighted-sum method, for one value of w. Conceptually, the weighted-sum method moves that line as far down and to the left as possible, revealing one point in the Pareto set.

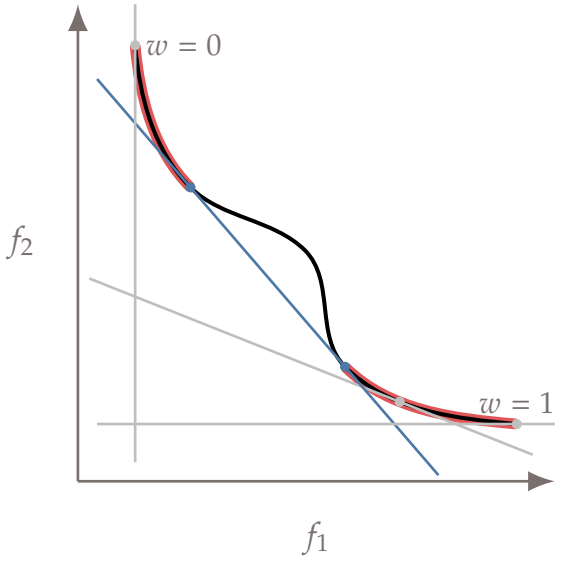


Figure 7.5: The convex portion of this Pareto front are the portions highlighted.

7.3.2 Epsilon Constraint

The epsilon-constraint method works by minimizing one objective, while setting all other objectives as additional constraints

$$\begin{aligned} \text{minimize} \quad & f_i \\ \text{by varying} \quad & x \\ \text{subject to} \quad & f_j \leq \epsilon_j \quad \text{for all } j \neq i, \\ & g(x) \leq 0, \\ & h(x) = 0. \end{aligned} \tag{7.7}$$

Then, we must repeat this procedure for different values of ϵ_j.

This method is visualized in Fig. 7.6. 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 ϵ.

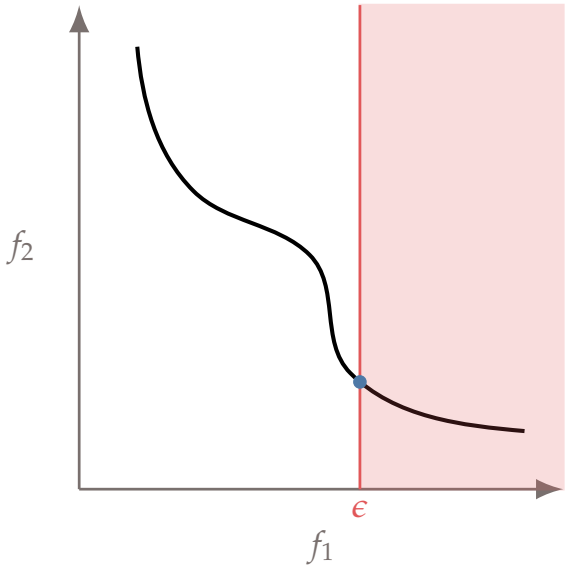


Figure 7.6: The vertical line represents an upper bound constraint on f_1. The other objective f_2 is minimized to reveal one point in the Pareto set. This procedure is then repeated for different constraints on f_1 to sweep out the Pareto set.

One advantage of this method is that determining appropriate values for ϵ 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 ϵ does not in general yield uniform spacing of the Pareto front and therefore it might may still be inefficient, particularly with more than two objectives.

7.3.3 Normal Boundary Intersection

The normal boundary intersection method addresses the issue of nonuniform spacing along the Pareto front. 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. Finally, we solve optimization problems that search along lines normal to this plane.

This procedure is shown in Fig. 7.7 for a two objective case. In this case, the plane that passes through the anchor points is a line. The shorter line is a constraint along which an optimal point is sought. The optimal point found along this line is shown on the Pareto front.

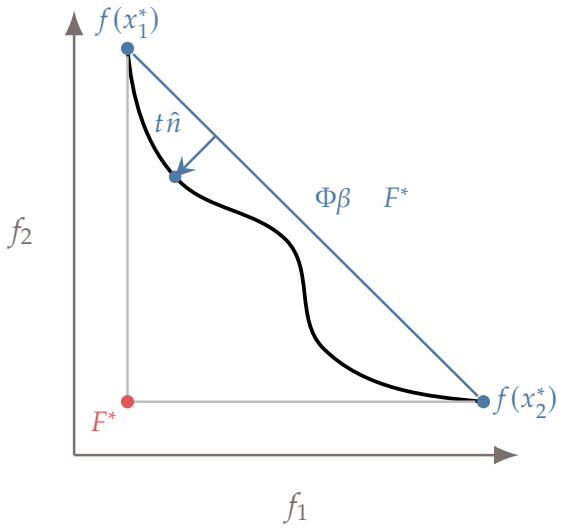


Figure 7.7: A notional example of the normal boundary intersection method. A plane is created passing through the single objective optima, and solutions are sought normal to that plane to allow for a more evenly spaced Pareto front.

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:

$$F^* = \begin{bmatrix} f_1(x_1^*) \\ f_2(x_2^*) \\ \vdots \\ f_n(x_n^*) \end{bmatrix}, \tag{7.8}$$

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,

$$\Phi\beta + F^*, \tag{7.9}$$

where the i^{th} column of Φ is $F(x_i^*) - F^*$ and the vector β is a weighting parameter that defines the plane. The entries are constrained such that $\beta_i \in [0, 1]$, and $\sum_i \beta_i = 1$. If we make $\beta_i = 1$ and all other entires zero, then this equation returns one of the anchor points, $F(x_i^*)$. The role of β is similar to the weighting parameter in the weighted-sum method. We assign different values of β, one at a time, to produce different points in the Pareto set.

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 t, yielding

$$\Phi\beta + F^* + t\hat{n}. \tag{7.10}$$

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,

$$\hat{n} = -\Phi\mathbf{1} \tag{7.11}$$

where $\mathbf{1}$ is a vector of ones.

We now solve the following optimization problem to yield a point on the Pareto front:

$$\begin{aligned} \text{maximize} \quad & t \\ \text{with respect to} \quad & x, t \\ \text{subject to} \quad & \Phi\beta + F^* + t\hat{n} = F(x) \\ & g(x) \leq 0 \\ & h(x) = 0 \end{aligned} \tag{7.12}$$

This means that we are finding the point furthest away from the anchor point plane (a point in the Pareto set), while satisfying the original problem constraints.

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 7.3. A two-dimensional normal boundary interface problem.

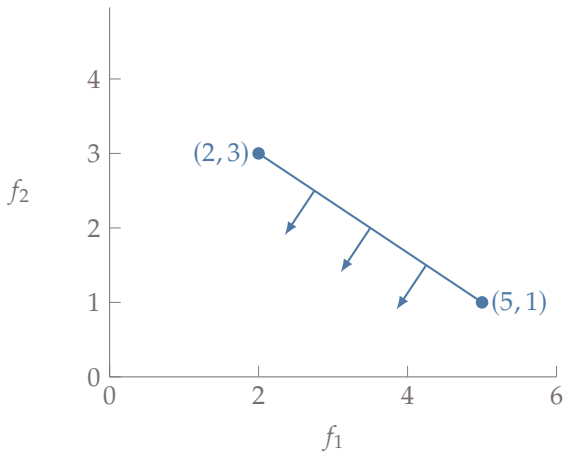


Figure 7.8: Search directions are normal to the line connecting anchor points.

First, we optimize the objectives one at a time, which in our example results in the two anchor points shown in Fig. 7.8: $F(x_1^*) = (2, 3)$ and $F(x_2^*) = (5, 1)$. The utopia point is then:

$$F^* = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{7.13}$$

For the matrix Φ recall that the i^{th} column of Φ is $F(x_i^*) - F^*$:

$$\Phi = \begin{bmatrix} 0 & 3 \\ 2 & 0 \end{bmatrix} \tag{7.14}$$

Our quasi-normal vector is given by $-\Phi\mathbf{1}$ (note that the true normal is $[-2, -3]$):

$$\hat{n} = \begin{bmatrix} -3 \\ -2 \end{bmatrix} \tag{7.15}$$

We now have all the parameters we need to solve Eq. (7.12). For two objectives, we would define β as $\beta = [w, 1 - w]^T$ and vary w in equal steps between 0 and 1.

7.3.4 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 6.4.

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 get something like the trend shown in Fig. 7.9. The red points are the current nondominated set. As the optimization progresses, the nondominated set moves down and to the left and eventually converges toward the true Pareto set.

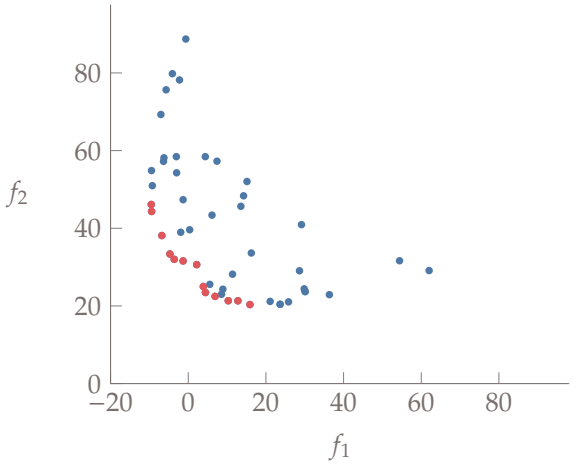


Figure 7.9: Population a multiobjective genetic algorithm iteration plotted against two objectives. The nondominated set is highlighted in red and converged toward the Pareto front.

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 a modern and popular approach, the NSGA-II algorithm.

After reproduction, both the parent generation and child 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. In NSGA-II, all points in the population are first sorted using the concept of *dominance depth*, which is also

called nondominated sorting. In this approach, all points in the population that are nondominated 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.

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.

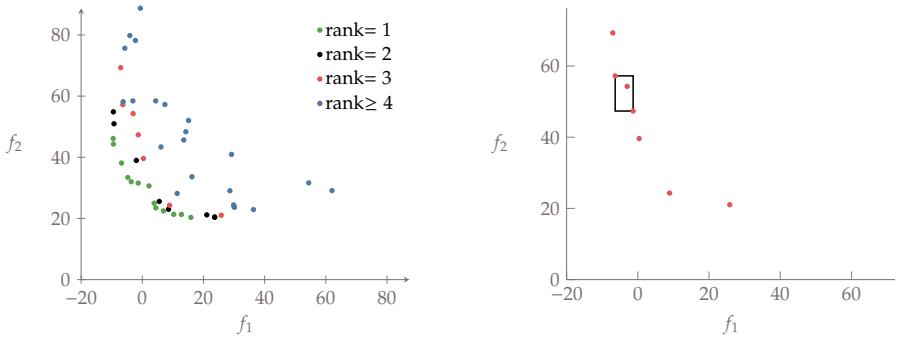
The way that we perform selection within a group that can only partially fit is to try to preserve diversity as much as possible. In NSGA-II, points in this last group are ordered by their *crowding distance*, which is a measure of how spread apart the points area. The algorithm seeks to preserve points that are well spread. For each point, the largest possible cuboid in objective space is formed around it. This cuboid just touches its neighboring points. The sum of the dimensions of this cuboid is the crowding distance. When summing the dimensions, each dimension is normalized by the maximum range of that objective value. In other words, 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:

$$\frac{f_{1_{i+1}} - f_{1_{i-1}}}{f_{1_N} - f_{1_1}} \tag{7.16}$$

The anchor points (the single objective optima) are assigned a crowding distance of infinity. Placement of points into the new population begins with the highest crowding distance and descends until the population size is filled.

For an example population, Fig. 7.10a highlights different ranks with different colors. Let us assume that all rank 1 and rank 2 points fit within the new population, but that all of rank 3 do not fit. We would then isolate the rank 3 points (Fig. 7.10b) and compute the crowding distance.

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.



(a) Points in the population highlighted by rank.

(b) A cuboid around one point demonstrating the definition of crowding distance (except that the distances are normalized).

Figure 7.10: Illustration of main concepts in NSGA-II

7.4 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 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.

7.5 Further Notes

- The constraint-epsilon method as first described by Haimes et al. [1].
- The normal boundary intersection method was proposed by Das and Dennis [2].

- 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 [3], 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 [4]. 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 [5].

- The first application of an evolutionary algorithm for solving a multiobjective problem was by Schaffer [6]. Various improvements and alternatives have been developed over the years. The NSGA-II algorithm was developed by Deb et al. [7]. Some key developments include using the concept of domination in the selection process, preserving diversity amongst the non-dominated set, and using elitism [8].

Bibliography

[1] Yacov Y. Haimes, Leon S. Lasdon, and David A. Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(3): 296–297, jul 1971. doi:10.1109/tsmc.1971.4308298.

[2] Indraneel Das and J. E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, aug 1998. doi:10.1137/s1052623496307510.

[3] A. Ismail-Yahaya and A. Messac. Effective generation of the pareto frontier using the normal constraint method. In *40th AIAA Aerospace Sciences Meeting & Exhibit*. American Institute of Aeronautics and Astronautics, jan 2002. doi:10.2514/6.2002-178.

[4] Achille Messac and Christopher A. Mattson. Normal constraint method with guarantee of even representation of complete pareto frontier. *AIAA Journal*, 42(10):2101–2111, oct 2004. doi:10.2514/1.8977.

[5] B. J. Hancock and C. A. Mattson. The smart normal constraint method for directly generating a smart pareto set. *Structural and Multidisciplinary Optimization*, 48(4):763–775, jun 2013. doi:10.1007/s00158-013-0925-6.

[6] James David Schaffer. *Some Experiments in Machine Learning Using Vector Evaluated Genetic Algorithms.* PhD thesis, Vanderbilt University, Nashville, TN, 1984.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002. doi:10.1109/4235.996017.

[8] Kalyanmoy Deb. Introduction to evolutionary multiobjective optimization. In *Multiobjective Optimization*, pages 59–96. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-88908-3_3.

CHAPTER 8

Multidisciplinary Design Optimization Architectures

As mentioned in Chapter 1, most engineering systems are multidisciplinary. In Section 2.4 we introduced the notion of multidisciplinary models. All the optimization methods covered so far apply to multidisciplinary problems, as long as the objective and constraints are computed through a multidisciplinary analysis at every optimization iteration. This approach is only one of the ways to solve and MDO problem.

In this chapter, we discuss other approaches—called *architectures*—for solving MDO problems that involve reformulating the original design optimization problem. We start with *monolithic* architectures, which involve a single optimization problem, and then introduce a few *distributed* architectures, which involve multiple optimization problems.

8.1 MDO Problem Representation

The MDO problem representation we use here is shown in Fig. 8.1 for a general three-component system. Here we use the functional representation introduced in Section 2.4.2, where the states in each component are hidden and we just see its output as a coupling variable at the system level.

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 $x = \left[x_0^T, x_1^T, \ldots, x_N^T\right]^T$.

The set of constraints is also split into global constraints and local ones.

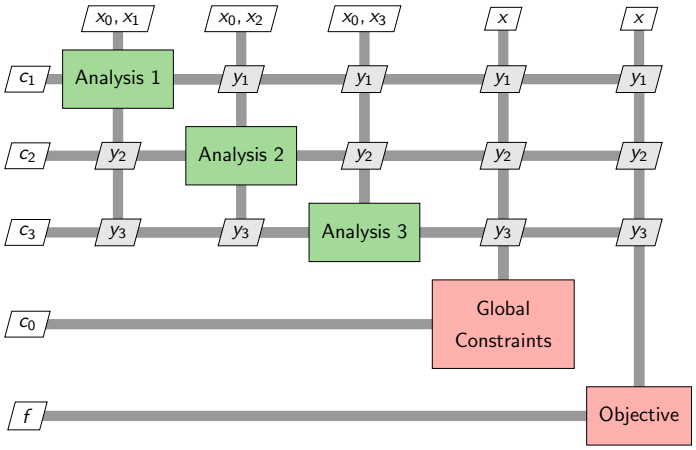


Figure 8.1: MDO problem nomenclature and dependencies.

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. 8.1.

8.2 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.

8.2.1 Multidisciplinary Feasible

The multidisciplinary design feasible (MDF) architecture is the closest to a single discipline problem because the design variables, objective, and constraints are exactly 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

$$\begin{aligned}
\text{minimize} \quad & f\left(x, y^*\right) \\
\text{with respect to} \quad & x \\
\text{subject to} \quad & c_0\left(x, y^*\right) \leq 0 \\
& c_i\left(x_0, x_i, y_i^*\right) \leq 0, \quad i = 1, \ldots, N. \\
\text{while solving} \quad & R_i\left(x, y\right) = 0 \qquad\quad i = 1, \ldots, N. \\
\text{with respect to} \quad & y
\end{aligned} \tag{8.1}$$

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 2.4.4. 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,

$$y_i = Y_i\left(x_0, x_i, y^*_{j\neq i}\right), \quad i = 1, \ldots, N, \tag{8.2}$$

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. 8.2. Here we use a Gauss–Seidel iteration to converge the MDA, but any other method for converging the MDA could be used.

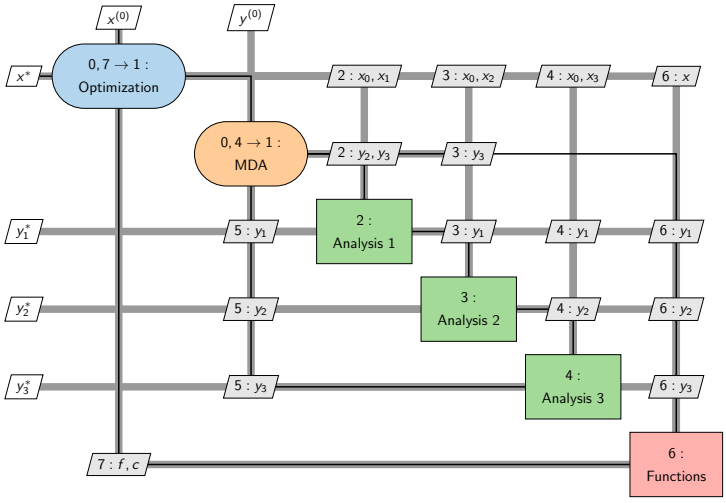


Figure 8.2: The MDF architecture relies on an MDA to solve for the coupling and state variables for each optimization iteration. In this case, the MDA uses a Gauss–Seidel approach.

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 4.10. Ideally, we would use one of the analytic coupled derivative computation methods of Section 4.10, which require a substantial implementation effort.

8.2.2 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.

The optimization problem for the IDF architecture is

$$\begin{aligned}
\text{minimize} \quad & f\left(x, y\right) \\
\text{with respect to} \quad & x, y^t \\
\text{subject to} \quad & c_0\left(x, y\right) \leq 0 \\
& c_i\left(x_0, x_i, y_i\right) \leq 0 \qquad i = 1, \ldots, N, \\
& c_i^c = y_i^t - y_i = 0 \qquad i = 1, \ldots, N, \\
\text{while solving} \quad & R_i\left(x, y_i, y_{j\neq j}^t\right) = 0 \qquad i = 1, \ldots, N. \\
\text{for} \quad & y
\end{aligned} \tag{8.3}$$

where each component is solved independently to compute the corresponding output coupling variables and constraints based on the target coupling variables, that is

$$\begin{aligned} y_i &= Y_i\left(x_0, x_i, y^t_{j\neq i}\right), \\ c_i &= c_i\left(x_0, x_i, y_i, y^t_{j\neq i}\right), \end{aligned} \tag{8.4}$$

where unlike MDF, we do not need to make the coupling variables consistent using an MDA, so each component only needs to be solved once per 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. 8.3.

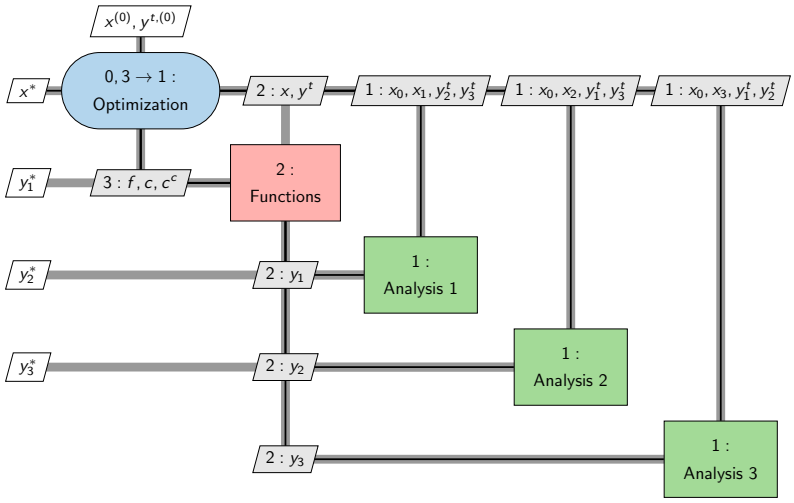


Figure 8.3: The IDF architecture breaks up the MDA by letting the optimizer solve for the coupling variables that satisfy interdisciplinary feasibility.

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 2.4.4.

8.2.3 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. 2.9.

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

$$
\begin{aligned}
\text{minimize} \quad & f_0\left(x, y\right) \\
\text{with respect to} \quad & x, y, u \\
\text{subject to} \quad & c_0\left(x, y\right) \leq 0 \\
& c_i\left(x_0, x_i, y_i\right) \leq 0 \qquad \text{for} \quad i = 1, \ldots, N \\
& \mathcal{R}_i\left(x_0, x_i, y, u_i\right) = 0 \quad \text{for} \quad i = 1, \ldots, N.
\end{aligned}
\tag{8.5}
$$

where we use the representation shown in Fig. 2.7, 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. 8.4

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

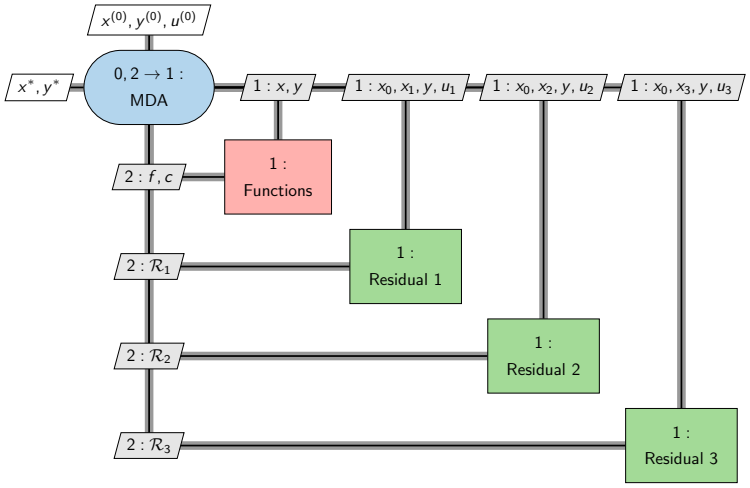


Figure 8.4: The SAND architecture lets the optimizer solve for all variables (design, coupling, and state variables) and component solvers are no longer needed.

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.

8.2.4 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 2.4.2. There are two main ideas in MAUD: 1) represent the coupled system as a single nonlinear system and 2) linearize the coupled system using the UDE (4.70) 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 $i = 1, \ldots, n$, as previously written in Eq. (2.20). 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. 2.8, 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 (2.20) as $\bar{u}$, the new state is,

$$u \equiv \begin{bmatrix} x \\ \bar{u} \\ f \end{bmatrix}. \tag{8.6}$$

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. (2.19). The complete set of residuals is then,

$$R(u) \equiv \begin{bmatrix} x - x_0 \\ -R_{\bar{u}}(x, \bar{u}) \\ f - F(x, \bar{u}) \end{bmatrix}, \tag{8.7}$$

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 (8.6) and residuals (8.7) 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.

$$\frac{\partial R}{\partial u} \frac{\mathrm{d}u}{\mathrm{d}r} = \mathcal{I}$$

4. **Reverse differentiation** (right equality of UDE): A linear system whose solution yields the derivative of one selected state 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 one associated with f.

$$\frac{\partial R}{\partial u}^T \frac{\mathrm{d}u}{\mathrm{d}r}^T = \mathcal{I}$$

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. 8.5. 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.

8.3 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.

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".

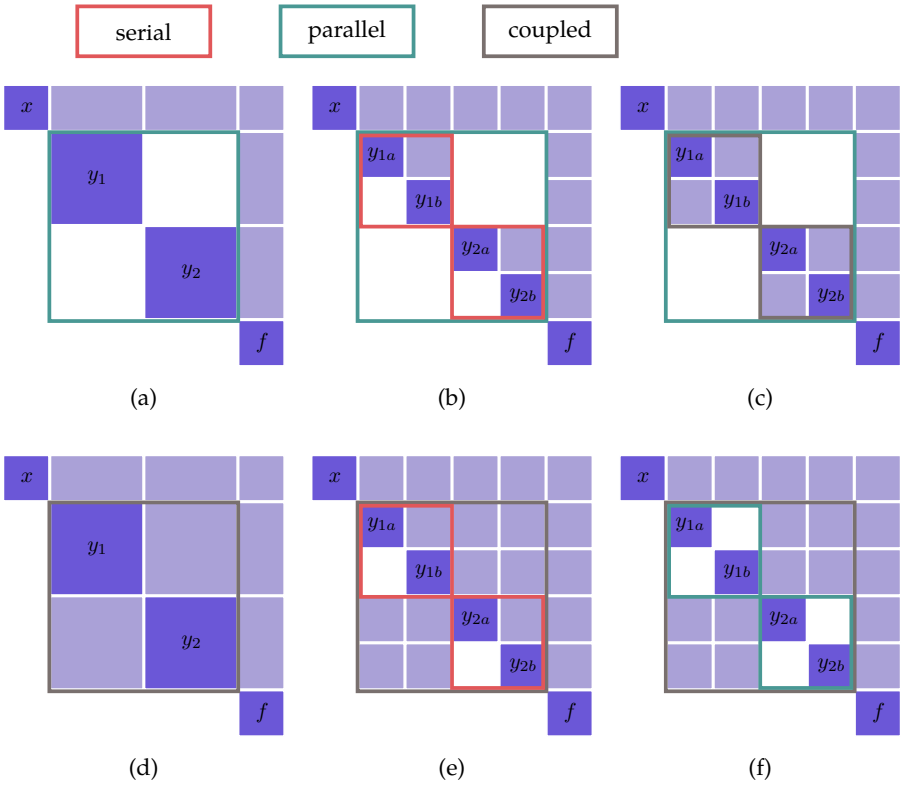


Figure 8.5: Example problem structures and corresponding MAUD hierarchical decompositions. The problem structure is shown using DSM. The hierarchical decompositions are shown above the matrices, where the components are shown in blue, serial groups in red, parallel groups in green, and coupled groups (i.e., those containing feedback loops) in gray. Each blue, red, green, or gray bar corresponds to an intermediate system in the numerical model.

8.3.1 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.

8.3.2 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. 8.6. 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

$$\begin{aligned}
\text{minimize} \quad & f_0\left(x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t\right) \\
\text{with respect to} \quad & x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t \\
\text{subject to} \quad & c_0\left(x_0, \hat{x}_1, \ldots, \hat{x}_N, y^t\right) \leq 0 \\
& J_i^* = ||\hat{x}_{0i} - x_0||_2^2 + ||\hat{x}_i - x_i||_2^2 + \\
& \qquad ||y_i^t - y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)||_2^2 = 0 \quad \text{for } i = 1, \ldots, N
\end{aligned} \tag{8.8}$$

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

$$\begin{aligned}
\text{minimize} \quad & J_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) \\
\text{with respect to} \quad & \hat{x}_{0i}, x_i \\
\text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) \leq 0.
\end{aligned} \tag{8.9}$$

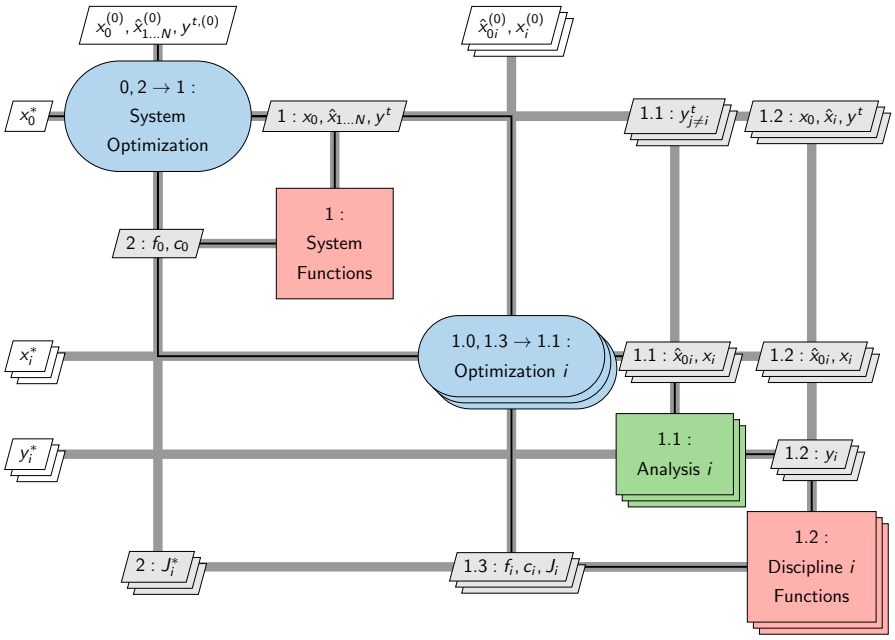


Figure 8.6: Diagram for the CO architecture.

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 Algorithm 22.

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.

8.3.3 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

Algorithm 22 Collaborative optimization

Input: Initial design variables x
Return: Optimal variables x^*, objective function f^*, and constraint values c^*
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 $\rightarrow$ 1.1: Optimization i has converged
end for
2: Compute a new system subproblem design point
until 2 $\rightarrow$ 1: System optimization has converged

ATC system-level problem is

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y^t\right) + \sum_{i=1}^{N} \Phi_i\left(\hat{x}_{0i} - x_0, y_i^t - y_i\left(x_0, x_i, y^t\right)\right) + \\ & \Phi_0\left(c_0\left(x, y^t\right)\right) \\ \text{with respect to} \quad & x_0, y^t, \end{aligned} \tag{8.10}$$

where Φ_0 is a penalty relaxation of the global design constraints and Φ_i is a penalty relaxation of the discipline i consistency constraints. The i^{th} discipline subproblem is:

$$\begin{aligned} \text{minimize} \quad & f_0\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right), y_{j\neq i}^t\right) + f_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) + \\ & \Phi_i\left(y_i^t - y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right), \hat{x}_{0i} - x_0\right) + \\ & \Phi_0\left(c_0\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right), y_{j\neq i}^t\right)\right) \\ \text{with respect to} \quad & \hat{x}_{0i}, x_i \\ \text{subject to} \quad & c_i\left(\hat{x}_{0i}, x_i, y_i\left(\hat{x}_{0i}, x_i, y_{j\neq i}^t\right)\right) \leq 0. \end{aligned} \tag{8.11}$$

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. 8.7 shows the ATC architecture XDSM, where w denotes the penalty function weights used in the determination of Φ_0 and Φ_i. The details of ATC are described in Algorithm 23.

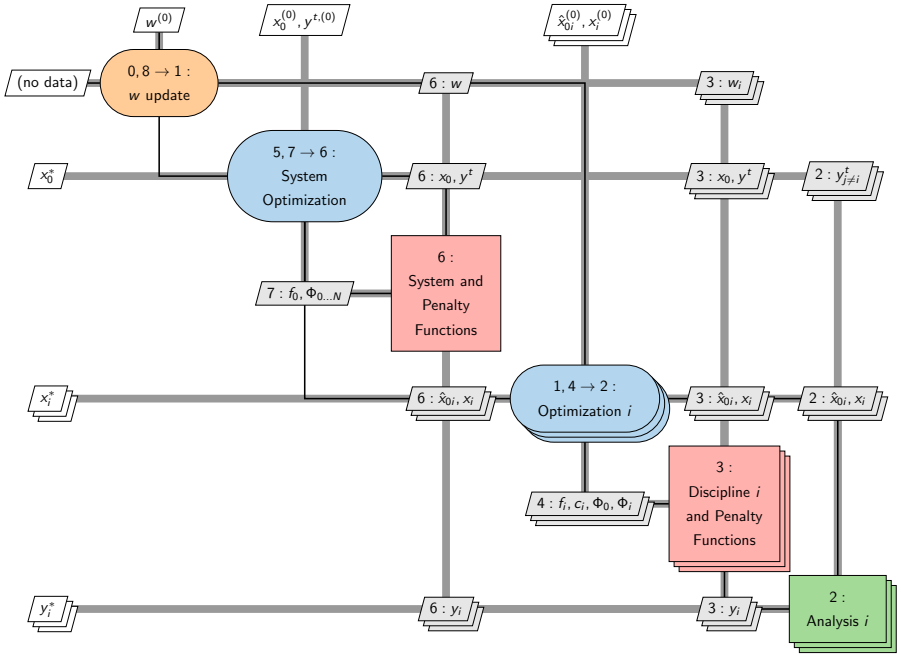


Figure 8.7: Diagram for the ATC architecture

Algorithm 23 Analytical target cascading

Input: Initial design variables x
Return: Optimal variables x^*, objective function f^*, and constraint values c^*

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

8.3.4 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 3.6. These approximations are constructed at each iteration using coupled derivatives (see Section 4.10). The system level subproblem is formulated as

$$\begin{aligned}
\text{minimize} \quad & (f_0^*)_0 + \left(\frac{\mathrm{d}f_0^*}{\mathrm{d}x_0}\right)\Delta x_0 \\
\text{with respect to} \quad & \Delta x_0 \\
\text{subject to} \quad & (c_0^*)_0 + \left(\frac{\mathrm{d}c_0^*}{\mathrm{d}x_0}\right)\Delta x_0 \leq 0 \\
& (c_i^*)_0 + \left(\frac{\mathrm{d}c_i^*}{\mathrm{d}x_0}\right)\Delta x_0 \leq 0 \quad \text{for} \quad i = 1, \ldots, N \\
& \Delta x_{0L} \leq \Delta x_0 \leq \Delta x_{0U}.
\end{aligned} \tag{8.12}$$

The discipline i subproblem is given by

$$\begin{aligned} \text{minimize} \quad & (f_0)_0 + \left(\frac{\mathrm{d}f_0}{\mathrm{d}x_i}\right)\Delta x_i \\ \text{with respect to} \quad & \Delta x_i \\ \text{subject to} \quad & (c_0)_0 + \left(\frac{\mathrm{d}c_0}{\mathrm{d}x_i}\right)\Delta x_i \leq 0 \\ & (c_i)_0 + \left(\frac{\mathrm{d}c_i}{\mathrm{d}x_i}\right)\Delta x_i \leq 0 \\ & \Delta x_{iL} \leq \Delta x_i \leq \Delta x_{iU}. \end{aligned} \tag{8.13}$$

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.

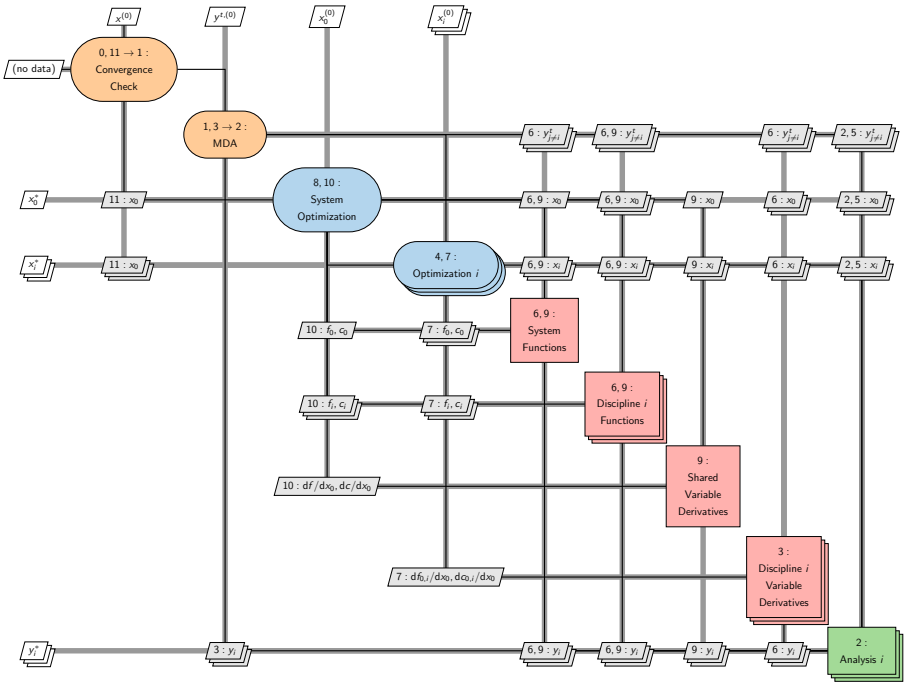


Figure 8.8: Diagram for the BLISS architecture

Algorithm 24 Bilevel integrated system synthesis

Input: Initial design variables x
Return: Optimal variables x^*, objective function f^*, and constraint values c^*
0: Initiate system optimization
repeat
1: Initiate MDA
repeat
2: Evaluate discipline analyses
3: Update coupling variables
until 3 $\rightarrow$ 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 $\rightarrow$ 1: System optimization has converged

Figure 8.8 shows the XDSM for BLISS and the corresponding steps are listed in Algorithm 24. 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.

8.3.5 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

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y\left(x, y\right)\right) + \sum_k f_k\left(x_0, x_k, y_k\left(x_0, x_k, y_{j\neq k}\right)\right) \\ \text{with respect to} \quad & x_0, x_k \\ \text{subject to} \quad & c_0\left(x, y\left(x, y\right)\right) \leq 0 \\ & c_k\left(x_0, x_k, y_k\left(x_0, x_k, y_{j\neq k}\right)\right) \leq 0 \quad \text{for all } k, \end{aligned} \tag{8.14}$$

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

$$\begin{aligned} \text{minimize} \quad & f_0\left(x, y\left(x, y\right)\right) + f_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}\right)\right) \\ \text{with respect to} \quad & x_i \\ \text{subject to} \quad & c_i\left(x_0, x_i, y_i\left(x_0, x_i, y_{j\neq i}\right)\right) \leq 0. \end{aligned} \tag{8.15}$$

Figure 8.9 shows a three-discipline case where the third discipline is replace with a design optimization. The corresponding sequence of operations in ASO is listed in Algorithm 25.

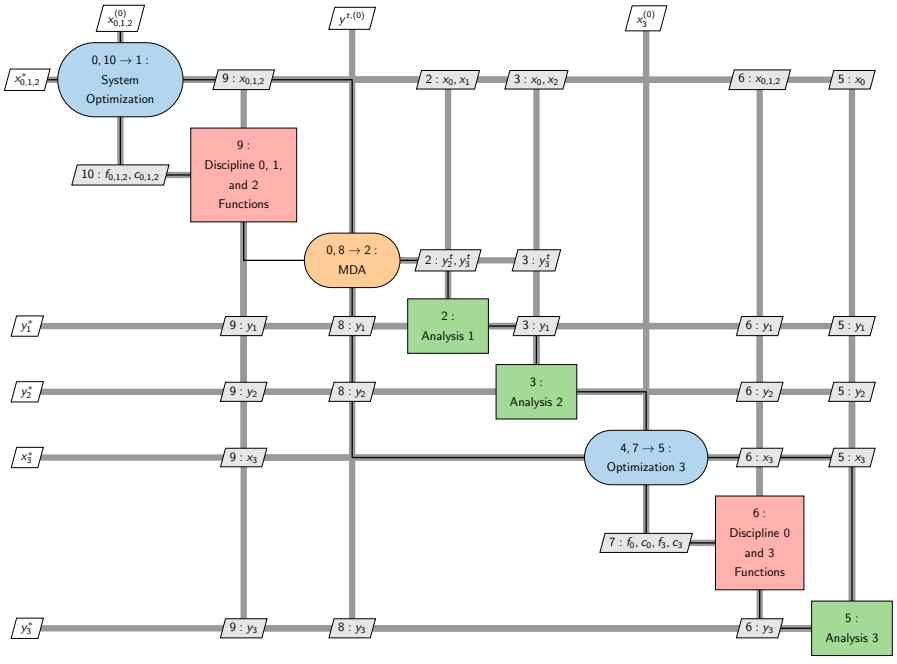


Figure 8.9: Diagram for the ASO architecture

Algorithm 25 ASO

Input: Initial design variables x
Return: Optimal variables x^*, objective function f^*, and constraint values c^*
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
6: Compute discipline 3 objectives and constraints
7: Update local design variables
until 7 $\rightarrow$ 5: Discipline 3 optimization has converged
8: Update coupling variables
until 8 $\rightarrow$ 2 MDA has converged
9: Compute objective and constraint function values for all disciplines 1 and 2
10: Update design variables
until 10 $\rightarrow$ 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.

8.3.6 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.

8.4 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. 8.10: 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.

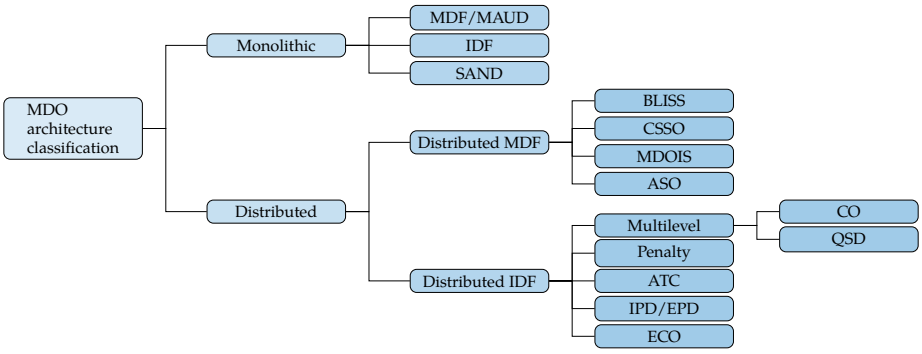


Figure 8.10: Classification of MDO architectures.

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 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). Fig. 8.10 includes other architectures not covered in this chapter; see Section 8.5 for more information.

8.5 Further Notes

- Kroo [1] describes many of the early challenges for large-scale MDO and some proposed solutions, with a focus on the CO architectures.
- Braun [2] showed that the CO problem statement is mathematically equivalent to the IDF problem statement (8.3) and, therefore, equivalent to the original MDO problem (8.1) as well. They formulated two versions of the CO architecture: CO_1 and CO_2. The version presented in Section 8.3.2 is CO_2.
- For the SAND architecture, the disciplinary residual equations could just be a set of discretized PDEs. This is the subject of the broad field know as *PDE-constrained optimization* [3].
- Martins and Lambe [4] present a comprehensive description of all MDO architectures, including references to known applications of each architecture.

- The performance of some of these MDO architectures is compared by Tedford and Martins [5] using a few benchmark problems, including a scalable one. They show that the monolithic architectures (MDF, IDF, and SAND) converged much faster than the two distributed ones that were implemented (CO and CSSO). In the last few years, the vast majority of MDO applications have used monolithic MDO architectures.

- The MAUD architecture was developed by Hwang and Martins [6], 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.

- MAUD was implemented in OpenMDAO V2 by Gray et al. [7]. OpenMDAO is an open-source framework developed by NASA to facilitate the development of multidisciplinary solvers.[1] 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. There are various applications developed using OpenMDAO that are publicly available, such as aerostructural wing design optimization [8], gas turbine analysis and optimization [9], and satellite design [10].

- There are a number of commercial MDO frameworks that are available, including Isight/SEE [11] 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 [12]. 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.

Bibliography

[1] Ilan M. Kroo. MDO for large-scale design. In N. Alexandrov and M. Y. Hussaini, editors, *Multidisciplinary Design Optimization: State-of-the-Art*, pages 22–44. SIAM, 1997.

[1] https://openmdao.org

[2] Robert D. Braun. *Collaborative Optimization: An Architecture for Large-Scale Distributed Design*. PhD thesis, Stanford University, Stanford, CA 94305, 1996.

[3] Lorenz T. Biegler, Omar Ghattas, Matthias Heinkenschloss, and Bart van Bloemen Waanders, editors. *Large-Scale PDE-Constrained Optimization*. Springer–Verlag, 2003.

[4] Joaquim R. R. A. Martins and Andrew B. Lambe. Multidisciplinary design optimization: A survey of architectures. *AIAA Journal*, 51(9):2049–2075, September 2013. doi:10.2514/1.J051895.

[5] Nathan P. Tedford and Joaquim R. R. A. Martins. Benchmarking multidisciplinary design optimization algorithms. *Optimization and Engineering*, 11 (1):159–183, February 2010. doi:10.1007/s11081-009-9082-6.

[6] John T. Hwang and Joaquim R. R. A. Martins. A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives. *ACM Transactions on Mathematical Software*, 44(4):Article 37, June 2018. doi:10.1145/3182393.

[7] Justin S. Gray, John T. Hwang, Joaquim R. R. A. Martins, Kenneth T. Moore, and Bret A. Naylor. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*, 59(4):1075–1104, April 2019. doi:10.1007/s00158-019-02211-z.

[8] John P. Jasa, John T. Hwang, and Joaquim R. R. A. Martins. Open-source coupled aerostructural optimization using Python. *Structural and Multidisciplinary Optimization*, 57(4):1815–1827, April 2018. doi:10.1007/s00158-018-1912-8.

[9] Eric S. Hendricks and Justin S. Gray. pyCycle: A tool for efficient optimization of gas turbine engine cycles. *Aerospace*, 6(87), August 2019. doi:10.3390/aerospace6080087.

[10] John T. Hwang, Dae Young Lee, James W. Cutler, and Joaquim R. R. A. Martins. Large-scale multidisciplinary optimization of a small satellite's design and operation. *Journal of Spacecraft and Rockets*, 51(5):1648–1663, September 2014. doi:10.2514/1.A32751.

[11] Oleg Golovidov, Srinivas Kodiyalam, Peter Marineau, Liping Wang, and Peter Rohl. Flexible implementation of approximation concepts in an MDO framework. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*. American Institute of Aeronautics and Astronautics, 1998. doi:10.2514/6.1998-4959.

[12] Vladimir Balabanov, Christophe Charpentier, D. K. Ghosh, Gary Quinn, Garret Vanderplaats, and Gerhard Venter. Visualdoc: A software system for general purpose integration and design optimization. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Atlanta, GA, 2002.

CHAPTER 9

Convex Optimization

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 and even for candidate problems that have the potential to be convex, significant experience is often needed to recognize and utilize techniques that will reformulate the problems into an appropriate form.

9.1 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. 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 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.[1]. A function f is convex if:

$$f((1-\eta)x_1 + \eta x_2) \leq (1-\eta)f(x_1) + \eta f(x_2) \tag{9.1}$$

for all x_1 and x_2 in the domain, where $0 \leq \eta \leq 1$. This requirement is illustrated in Fig. 9.1 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.

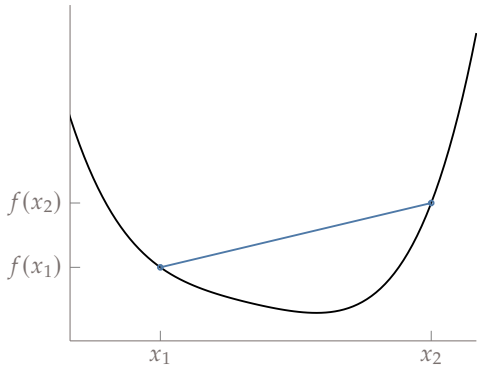


Figure 9.1: Illustration of what it means for a function to be convex in the one-dimensional case. The function (blue) must be below a line that connects any two points (red) in the domain.

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 options to take advantage of convexity. The first one is to directly formulate the problem in a known convex form. The second option is to use a *disciplined convex problem*, 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

[1]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.

great simplified, and thus reduced in fidelity. 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 a 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, for example in controls or machine learning, domains in which convex optimization is heavily used. In these cases, simplifying by performing a local linearization, for example, is less problematic 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 is infrequently used for design applications, with the exception of some limited uses of geometric programming, a topic discussed in more detail in a subsection below.

This chapter is introductory in nature, focusing only on understanding what convex optimization is useful for and what some of the most widely used forms are.

9.2 Convex Optimization Problems

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. The first two categories are well known and appear in a wide variety of applications (including in other chapters in this book). They are the only two forms we discuss in detail in this chapter, and many solvers exist for these types of problems.

Convex optimization problems are usually not directly expressed in these latter forms. 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 discussed in Section 9.2.3.

Finally, we introduce geometric programming. Geometric programming problems are actually not convex, but with a change of variables can be transformed to an equivalent convex form.

9.2.1 Linear Programming

A *linear program* (LP) has a linear objective and linear constraints and can be written as

$$\begin{aligned} \text{minimize} \quad & d^T x \\ \text{subject to} \quad & Ax + b = 0 \\ & \hat{A}x + \hat{b} \leq 0, \end{aligned} \tag{9.2}$$

All LPs are convex.

Example 9.1. 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 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:

$$\text{minimize} \quad \sum_j c_j x_j = c^T x \tag{9.3}$$

To meet the nutritional requirement of nutrient j we need to satisfy:

$$\sum_j n_{ij} x_j \geq R_i \Rightarrow Nx \geq R. \tag{9.4}$$

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. 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.

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 Example 9.1, 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 does 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 10.

9.2.2 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:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T Q x + d^T x \\ \text{subject to} \quad & Ax + b = 0 \\ & \hat{A}x + \hat{b} \leq 0 \end{aligned} \tag{9.5}$$

A QP is only convex *if* the matrix Q is positive semidefinite. A QP is reduced to an LP if $Q = 0$.

Example 9.2. The least-squares regression QP.

One of the most common QP examples is the 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:

$$\text{minimize} \quad \sum_i r_i^2 = \sum_i (\hat{b}_i - b_i)^2, \tag{9.6}$$

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 wish to optimize to fit the data. Note that 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:

$$f(\zeta) = c_1\zeta^2 + c_2\zeta + c_3 \tag{9.7}$$

This equation is linear in the coefficients (which corresponds to x above). 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:

$$\text{minimize} \quad \sum_i (a_i^T x - b_i)^2, \tag{9.8}$$

where a_i^T is the i^{th} row in the matrix A. Equivalently, we can express the least-squares problem as minimizing the square of a 2-norm:

$$\text{minimize} \quad ||Ax - b||_2^2 \tag{9.9}$$

which can be expressed equivalently as

$$||Ax - b||_2^2 = (Ax - b)^T(Ax - b) = x^T A^T Ax - 2b^T Ax + b^T b \tag{9.10}$$

This is the same as the general QP form, where $Q = 2A^T A$, $d = -2A^T b$, 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, 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 9.3. Linear-quadratic regulator (LQR) controller.

Another common example of a QP occurs in optimal control. Consider a discrete-time linear dynamic system:

$$x_{t+1} = Ax_t + Bu_t \tag{9.11}$$

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:

$$\text{minimize} \quad \frac{1}{2}\sum_{t=0}^{N}\left(x_t^T Q x_t + u_t^T R u_t\right), \tag{9.12}$$

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 7) 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.

A problem related to a QP is a *quadratically-constrained quadratic program* (QCQP). This is the same as a QP with the addition of quadratic inequality constraints, that is,

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}x^T Q x + d^T x \\ \text{subject to} \quad & \frac{1}{2}x^T R_i x + a_i^T x + b_i \leq 0 \text{ for } i = 1, \ldots, m \\ & \hat{A}x + \hat{b} = 0. \end{aligned} \tag{9.13}$$

Both Q and R must be positive semidefinite for the QCQP to be convex. A QCQP reduces to a QP if $R = 0$. A QCQP occurs in solving trust-region problems, which we discussed in Section 3.6, although for trust-region problems only an approximate solution method is typically used.

9.2.3 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.

The following are examples of functions that are convex, note that some of the functions are not even continuously differentiable as that is not a requirement of convexity:

- Exponential functions: $\exp^{ax}$ 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: $\max(x_1, x_2, \ldots, x_n)$
- log-sum-exp: $\log(\exp^{x_1} + \exp^{x_2} + \ldots + \exp^{x_n})$

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

- 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.

9.2.4 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:

$$f(x) = c x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n} \tag{9.14}$$

where $c > 0$ and all $x_i > 0$. A posynomial is a sum of monomials:

$$f(x) = \sum_{j=1}^{N} c_j x_1^{a_{1j}} x_2^{a_{2j}} \cdots x_n^{a_{nj}} \tag{9.15}$$

where all $c_j > 0$.

Example 9.4. 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:

$$L = C_L \frac{1}{2} \rho V^2 S \tag{9.16}$$

Total incompressible drag, a sum of parasitic and induced drag, is a posynomial:

$$D = C_{D_p} q S + \frac{C_L^2}{\pi A R e} q S \tag{9.17}$$

A GP in standard form is written as:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq 1 \\ & h_i(x) = 1 \end{aligned} \tag{9.18}$$

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:

$$\begin{aligned} \text{minimize} \quad & \log f_0(x) \\ \text{subject to} \quad & \log f_i(x) \leq 0 \\ & \log h_i(x) = 0. \end{aligned} \tag{9.19}$$

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:

$$\log(c x_1^{a_1} x_2^{a_2} \dots x_n^{a_n}) = 0 \tag{9.20}$$

Using the properties of logarithms, this can be expanded to an equivalent expression:

$$\log c + a_1 \log x_1 + a_2 \log x_2 + \ldots + a_n \log x_n = 0 \tag{9.21}$$

Introducing a change of variables, $y_i = \log x_i$, results in the following equality constraint:

$$\begin{aligned} a_1 y_1 + a_2 y_2 + \ldots + a_n y_n + \log c &= 0 \\ a^T y + \log c &= 0 \end{aligned} \tag{9.22}$$

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:

$$\log\left(\sum_{j=1}^{N} c_j x_1^{a_{1j}} x_2^{a_{2j}} \dots x_n^{a_{nj}}\right) \tag{9.23}$$

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 $x_i = \exp^{y_i}$):

$$\begin{aligned} \log f_i &= \log\left(\sum_{j=1}^{N} c_j \exp^{y_1 a_{1j}} \exp^{y_2 a_{2j}} \dots \exp^{y_n a_{nj}}\right) \\ &= \log\left(\sum_{j=1}^{N} c_j \exp^{y_1 a_{1j} + y_2 a_{2j} + y_n a_{nj}}\right) \\ &= \log\left(\sum_{j=1}^{N} \exp^{a_j^T y + b_j}\right) \text{ where } b_j = \log c_j. \end{aligned} \tag{9.24}$$

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 Example 9.4 [1].

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 [2], 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 of design problems that can be solved using geometric programing techniques [3, 4].

9.3 Further Notes

- The textbook by Boyd and Vandenberghe [5] is a good starting point for those seeking a deeper dive into the field of convex optimization.
- Agrawal et al. [6] provide a more detailed classification of the various types of convex
- Grant et al. [7] shows how the disciplined convex problem rules always translates the problem into an efficiently solvable form automatically.
- Stephen Boyd's research group has developed some freely available and popular tools for disciplined convex programming. The CVX family includes: CVX (Matlab),[2] CVXPY (Python),[3] Convex.jl (Julia),[4] and CVXR

[2] http://cvxr.com/cvx/
[3] http://www.cvxpy.org
[4] https://convexjl.readthedocs.io

(R).[5]

- A good introduction to geometric programming can be found in the paper by Boyd et al. [8] and in an introductory video by Hoburg.[6] Hoburg has also created a modeling language to make it easier to setup, solve, and explore GPs.[7]
- Many good references discuss other convex formulations like semidefinite programming and cone programming, complete with many examples [9, 10, 11, 12].

Bibliography

[1] Warren Hoburg and Pieter Abbeel. Geometric programming for aircraft design optimization. *AIAA Journal*, 52(11):2414–2426, nov 2014. doi:10.2514/1.j052732.

[2] Warren Hoburg, Philippe Kirschen, and Pieter Abbeel. Data fitting with geometric-programming-compatible softmax functions. *Optimization and Engineering*, 17(4):897–918, aug 2016. doi:10.1007/s11081-016-9332-3.

[3] Philippe G. Kirschen, Martin A. York, Berk Ozturk, and Warren W. Hoburg. Application of signomial programming to aircraft design. *Journal of Aircraft*, 55(3):965–987, may 2018. doi:10.2514/1.c034378.

[4] Martin A. York, Warren W. Hoburg, and Mark Drela. Turbofan engine sizing and tradeoff analysis via signomial programming. *Journal of Aircraft*, 55(3):988–1003, may 2018. doi:10.2514/1.c034463.

[5] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, Mar 2004. ISBN 0521833787.

[6] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, jan 2018. doi:10.1080/23307706.2017.1397554.

[7] Michael Grant, Stephen Boyd, and Yinyu Ye. Disciplined convex programming. In *Global Optimization*, pages 155–210. Kluwer Academic Publishers, 2006. doi:10.1007/0-387-30528-9_7.

[8] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 8(1): 67–127, apr 2007. doi:10.1007/s11081-007-9001-7.

[5] http://cvxr.rbind.io
[6] https://www.nas.nasa.gov/publications/ams/2015/06-05-15.html
[7] http://gpkit.readthedocs.io

[9] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. Applications of second-order cone programming. *Linear Algebra and its Applications*, 284(1-3):193–228, nov 1998. doi:10.1016/s0024-3795(98)10032-0.

[10] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, mar 1996. doi:10.1137/1038003.

[11] Lieven Vandenberghe and Stephen Boyd. Applications of semidefinite programming. *Applied Numerical Mathematics*, 29(3):283–299, mar 1999. doi:10.1016/s0168-9274(98)00098-1.

[12] Neal Parikh and Stephen Boyd. Block splitting for distributed optimization. *Mathematical Programming Computation*, 6(1):77–102, oct 2013. doi:10.1007/s12532-013-0061-8.

CHAPTER 10

Discrete Optimization

Most algorithms in this book have assumed that design variables are continuous. However, sometimes design variables must be discrete. Common examples of discrete optimization include scheduling, network problems, resource allocation, etc. This chapter introduces some techniques for dealing with discrete optimization problems.

10.1 Integer Programming

Discrete optimization can be classified with three different labels: binary, 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 doesn't make sense to design 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*. Many problems have both continuous and discrete variables, and these problems are referred to as *mixed integer programming*.

Unfortunately integer programming is NP-complete. Basically, this 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 10.1. The drawback of an exhaustive search.

The scaling difficulty is illustrated by a well-known integer programming problem: the traveling salesman problem. Consider a set of cities represented graphically on the left of Example 10.1. The problem is to find the shortest possible route that visits each city exactly once and returns to the starting city. Example 10.1 envisions one such solution (probably not 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 sustainable 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.

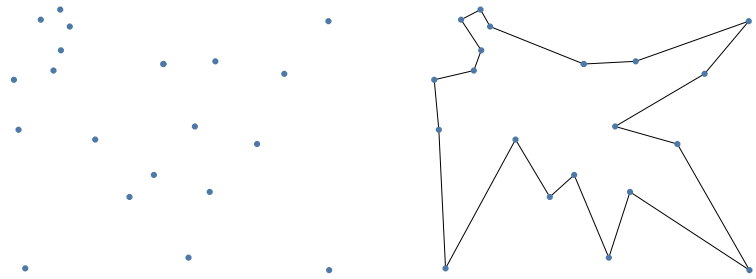


Figure 10.1: An example instance of the traveling salesman problem.

The apparent advantage of a discrete optimization problem is that we can construct algorithms that will find the global algorithm—exhaustive search for example. Exhaustive search ideas can also be used for continuous problems (see Section 6.6 for example, but the cost is much higher). The downside is that while an algorithm may eventually arrive at the right answer, as Example 10.1 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.

10.2 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 much more difficult and inefficient as 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 10.2.** 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 is going to fall within a reasonably small set (e.g., 2–6). Assuming there are no other discrete variable then we could just repeat the optimization five times 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 afterwards. This is usually justifiable if the magnitude of the design variables is large, and/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.

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.

10.3 Greedy Algorithms

Greedy algorithms are perhaps the simplest approach. 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.

Even for a fixed problem there is, 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 [1]), 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.

Example 10.3. Some greedy algorithms.

Here are some examples of greedy algorithms for different problems.

- Traveling salesman (Example 10.1): Always select the nearest city as the next step.
- Propeller problem (Example 10.2 but 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 (Example 9.1)[a]: 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.

[a] This is a form of the knapsack problem, which is a classic problem in discrete optimization

10.4 Branch and Bound

A popular method to solve integer programming problems is the *branch and bound* method. This approach is particularly effective with *linear* integer programing problems (the objective and constraints are all linear). It can be extended to nonlinear integer programming problems, but is generally far less effective. In this section we will assume linear mixed integer problems. Mathematically a linear mixed integer programming problem can be expressed as:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & \hat{A}x \leq \hat{b} \\ & Ax + b = 0 \\ & x_i \in \mathbb{Z}^+ \text{ for some or all i} \end{aligned} \tag{10.1}$$

Note that $\mathbb{Z}$, Zahlen, is a standard symbol for the set of all integers, where as $\mathbb{Z}^+$ means the set of all positive integers.

10.4.1 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.

Example 10.4. Converting an integer problem to a binary one.

Consider a problem where an engineering device may use one of n different materials: $y \in (1 \ldots n)$. 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:

$$\sum_{i=1}^{n} x_i = 1 \tag{10.2}$$

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 9.2.1). 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 10.2 shows a simplified depiction of 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 can't 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 can't exist on that branch. The algorithm keeps track of the best

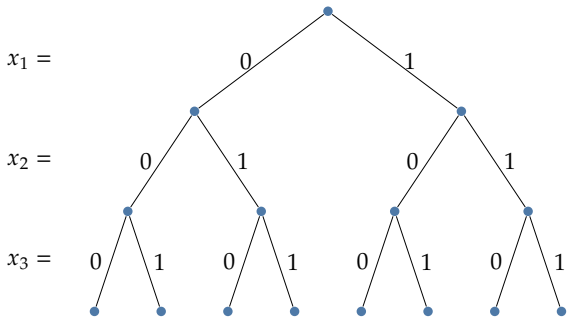


Figure 10.2: Enumerating the options for a binary problem with branching.

solution to the problem found so far. If one of the relaxed problems returns an objective that is worse then the best we've 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 Algorithm 26.

Algorithm 26 Branch and bound algorithm.

```
Initialize best solution found so far: f_best = ∞
Let S be the set of indices for binary constrained design variables
while branches remain do
    Solve relaxed problem for x̂, f̂
    if relaxed problem is infeasible then
        prune this branch, back up tree
    else
        if x̂_i ∈ {0, 1} ∀ i ∈ S then                ▷ A solution is found
            f_best = min(f_best, f̂), back up tree
        else
            if f̂ > f_best then
                prune this branch, back up tree
            else                                    ▷ A better solution might exist.
                branch further
            end if
        end if
    end if
end while
```

Return: optimal point $x^* = x_{best}$ and corresponding function value $f^* = f_{best}$

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:

$$\min_i |x_i - 0.5| \tag{10.3}$$

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 won't 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.

Example 10.5. A binary branch and bound optimization.

Consider the following problem:

$$\begin{aligned} \text{minimize} \quad & -2.5x_1 - 1.1x_2 - 0.9x_3 - 1.5x_4 \\ \text{subject to} \quad & 4.3x_1 + 3.8x_2 + 1.6x_3 + 2.1x_4 \leq 9.2 \\ & 4x_1 + 2x_2 + 1.9x_3 + 3x_4 \leq 9 \\ & x_i \in \{0, 1\} \text{ for all } i \end{aligned} \tag{10.4}$$

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:

$$\begin{aligned} x^* &= [1, 0.5274, 0.4975, 1]^T \\ f^* &= -5.0279 \end{aligned} \tag{10.5}$$

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. 10.3).

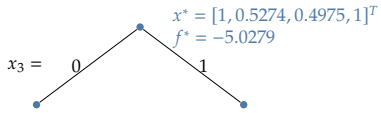


Figure 10.3: Initial binary branch.

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. 10.4. 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).

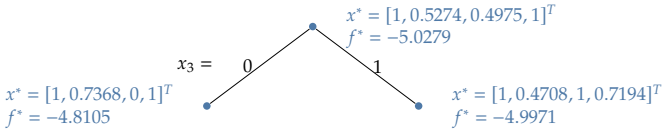


Figure 10.4: Solutions along these two branches.

The first branch (see Fig. 10.5) yields a feasible binary solution! The corresponding function value $f = -4$ is saved as the best value we've seen so far. There is no need to continue on this branch as the solution cannot be improved on this particular branch.

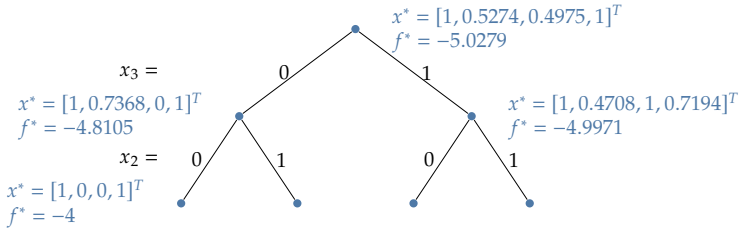


Figure 10.5: The first feasible solution.

We continue solving along the rest of this row (Fig. 10.6). 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've 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've found so far. Thus, we can prune both of these branches.

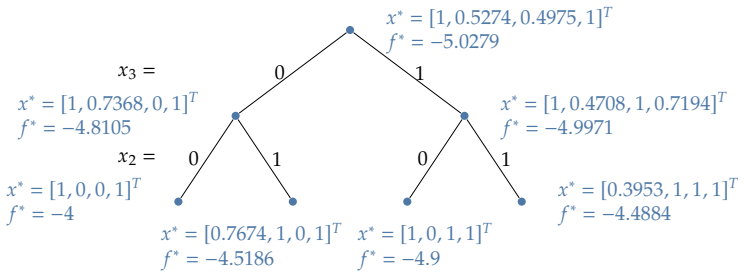


Figure 10.6: The rest of the solutions on this row.

All branches have been pruned and so we have solved the original problem:

$$\begin{aligned} x^* &= [1, 0, 1, 1]^T \\ f^* &= -4.9 \end{aligned} \tag{10.6}$$

Alternatively, we could have used a depth-first strategy. In this case, it is less efficient, but in general that isn't known beforehand. The depth-first tree for this same example is depicted in Fig. 10.7. Feasible solutions to the problem are shown with f^*.

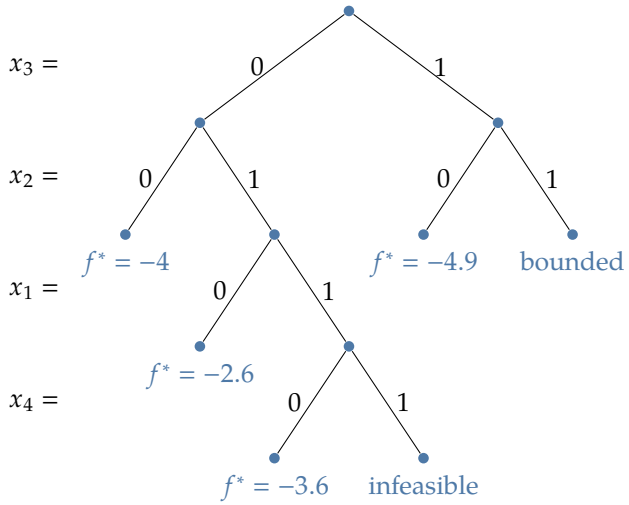


Figure 10.7: The search path with a depth-first strategy instead.

10.4.2 Integer Variables

If the problem can't 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 problem with the constraints: $x_i \leq 3$ or $x_i \geq 4$. An example of this is shown below.

Example 10.6. Branch and bound with integer variables.

Consider the following problem:

$$\begin{aligned} \text{minimize} \quad & -x_1 - 2x_2 - 3x_3 - 1.5x_4 \\ \text{subject to} \quad & x_1 + x_2 + 2x_3 + 2x_4 \leq 10 \\ & 7x_1 + 8x_2 + 5x_3 + x_4 = 31.5 \\ & x_i \in \mathbb{Z}^+ \text{ for } i = 1, 2, 3 \\ & x_4 \geq 0 \end{aligned} \tag{10.7}$$

We begin by solving the LP relaxation, in order words we remove the integer constraints, but with a lower bound of 0. The solution to that problem is:

$$x^* = [0, 1.1818, 4.4091, 0], f^* = -15.59 \tag{10.8}$$

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:

$$x^* = [0, 1.4, 4, 0.3], f^* = -15.25 \tag{10.9}$$

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. Rather than writing out all the details, the breadth-first tree is shown in Fig. 10.8. The figure gives some indication why integer problems are generally less efficient than binary ones. Note that 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 $x_3 \geq 4$. 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.

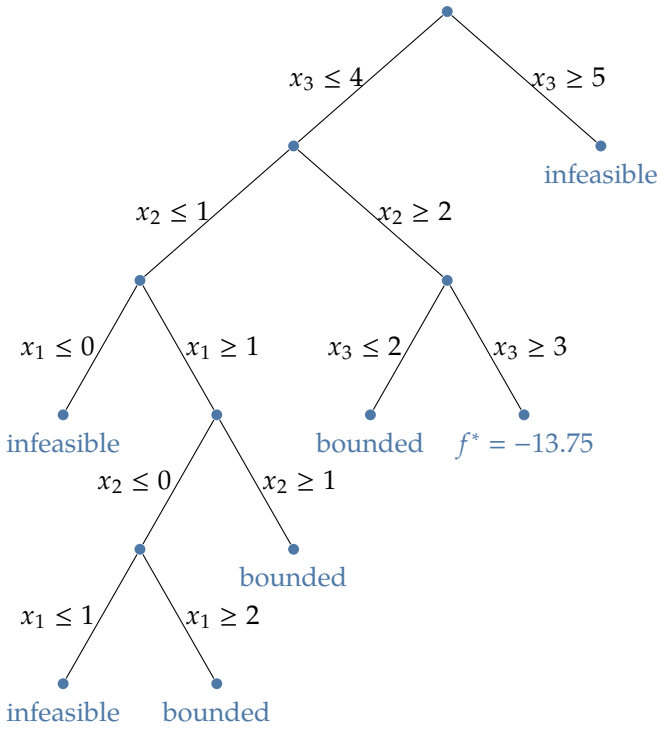


Figure 10.8: A breadth-search of the mixed integer programming example.

Once all the branches are pruned we see that the solution is:

$$\begin{aligned} x^* &= [0, 2, 3, 0.5]^T \\ f^* &= -13.75. \end{aligned} \tag{10.10}$$

10.5 Gradient-free Methods

Some of the gradient-free methods we discussed in Chapter 6 can be applied to discrete variables. In particular, the binary form the GA (Section 6.4.1) can be directly used with discrete variables. Since we use a discrete representation for the members of the population anyway, using discrete design variables is a natural fit. If a problem can be put in a linear form (excepting the discrete constraints) then the branch and bound methods will generally be much more efficient. However, if the problem is fundamentally nonlinear in the design variables, then a binary GA may be a good choice.

Another gradient-free method that is commonly used with discrete problems is simulated annealing. The idea in inspired by the annealing process of

materials. The atoms/molecules in a structure form a crystal lattice structure. If the material is heated then cooled back down the structure recrystallizes in a different minimum energy state. Simulated annealing is inspired by that process and uses concepts from statistical mechanics to mathematically model an annealing process where the optimization problem moves to a new minimum energy (objective) state. The details of the algorithm are not elaborated on here, but can be found in the literature [2]. Simulated annealing is not population-based like an evolutionary algorithm.

10.6 Further Notes

- Dynamic programming is another useful discrete optimization approach but is more specialized than the ones discussed in this chapter. Such as problem must have the *Markov property*, which is that future states depend only on a current state and not past states. If this is the case 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 if only one item existed in the store, then two items, then three items, each reusing the previous solutions (on the surface this may sound like a greedy optimization, but it is not). This approach has become particularly useful in some areas of economics, computational biology, and controls. More general design problems, like the propeller problem discussion in this chapter, don't fit this type of structure (i.e., choosing the number of blades can't be broken up into a smaller problem separated from choosing the material).
- This chapter only attempts to provide a broad overview of discrete optimization. For those interested in discrete methods, dedicated textbooks on the subject are recommended.

Bibliography

[1] Gregory Gutin, Anders Yeo, and Alexey Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the tsp. *Discrete Applied Mathematics*, 117(1-3):81–86, Mar 2002. ISSN 0166-218X. doi:10.1016/s0166-218x(01)00195-0.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983. ISSN 1095-9203. doi:10.1126/science.220.4598.671.

Exercises

1. *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 are available for use. We would like to 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.

2. *Branch and bound process.* Start with the problem defined in Example 10.5. Without looking at the example, go through the branch and bound procedure by solving a sequence of linear programming problems. Refer to the example to check your answer.

3. *Solve an integer linear programming problem.* Refer to the linear programing problem defined here. Solve the problem as a continuous problem then solve it again where the number of pallettes must be an integer.

CHAPTER 11

Optimization Under Uncertainty

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 uncertainity* (OUU) is the optimization of systems in the presence of uncertain parameters or design variables.

11.1 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 variation in the random design variables and parameters.[1]

Example 11.1. Robust versus reliable designs.

A familiar example of robust design occurs in playing the board game

[1] While we maintain a distinction in this book, many papers in the literature refer to both concepts as robust optimization.

> 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: your 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 isn't free and instead requires a tradeoff in peak performance. This is known as a risk-reward tradeoff.
> A familiar example of reliable design is 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 an actively growing field.

11.2 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, etc. 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 like: bar length, bar diameter, material Young's modulus, etc.

One measurement doesn't 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:

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{11.1}$$

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[2] estimate of the variance is:

$$\sigma_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2 \tag{11.2}$$

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:

$$\sigma_x^2 = \frac{1}{N-1}\left(\sum_{i=1}^{N}\left(x_i^2\right) - N\mu_x^2\right) \tag{11.3}$$

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. 11.1a. 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:

$$\text{Prob}[a \leq x \leq b] = \int_a^b p(x)dx \tag{11.4}$$

The total integral of the PDF must be one since it contains all possible outcomes (100%).

$$\int_{-\infty}^{\infty} p(x)dx = 1 \tag{11.5}$$

[2]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.

From the PDF we can also measure various statistics like the mean:

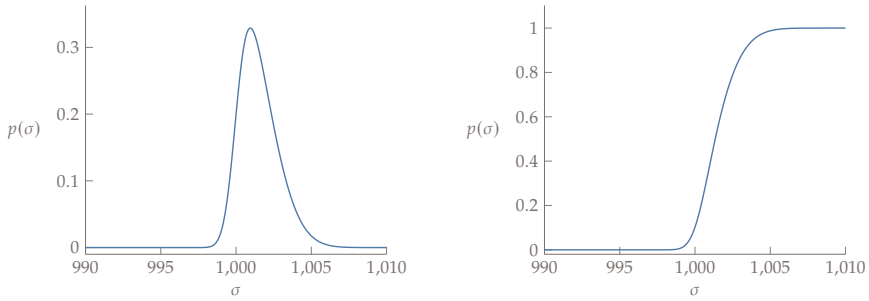
$$\mu_x = E[x] = \int_{-\infty}^{\infty} x p(x) dx \tag{11.6}$$

This quantity is also referred to as the expected value of x ($E[x]$). We can also compute the variance from its definition:

$$\sigma_x^2 = \int_{-\infty}^{\infty} (x - \mu_x)^2 p(x) dx \tag{11.7}$$

or in a mathematically equivalent format:

$$\sigma_x^2 = \int_{-\infty}^{\infty} x^2 p(x) dx - \mu_x^2 \tag{11.8}$$



(a) Probability density function for the axial strength of a rod.

(b) Cumulative distribution function for the axial strength of a rod.

Figure 11.1: 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:

$$F(x) = \int_{-\infty}^{x} f(t) dt \tag{11.9}$$

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. 11.1b. The CDF always approaches 1 as $x \to \infty$.

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:

$$p(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x - \mu)^2}{2\sigma^2} \tag{11.10}$$

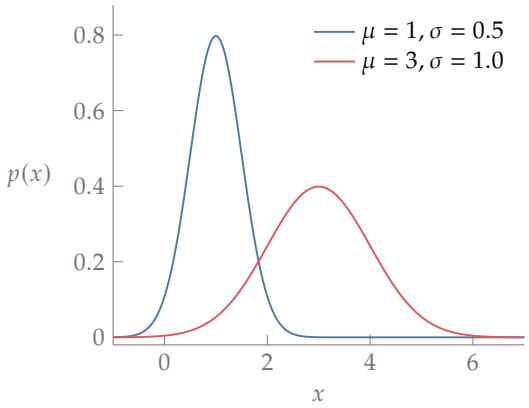
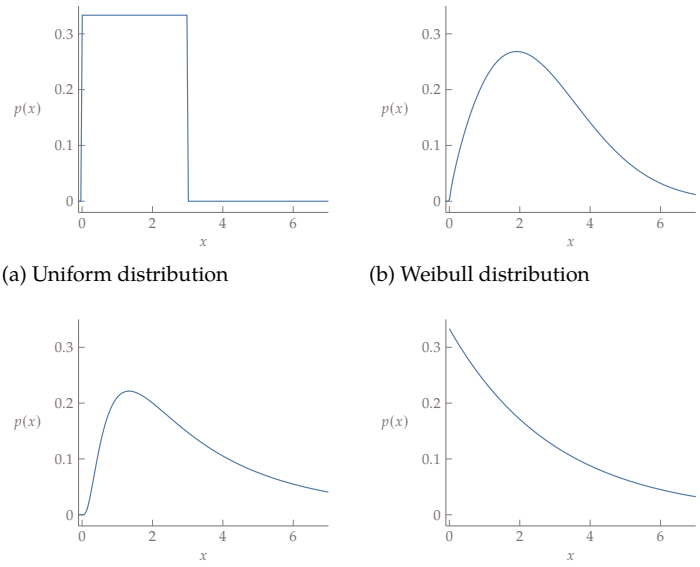


Figure 11.2: Two normal distributions. Changing the mean causes a shift along the x-axis. Increasing the standard deviation causes the PDF to spread out.

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 11.2 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. 11.3. These only give a flavor of different named distributions, many others exist.

11.3 Robust Design

We illustrate some of the key concepts in robust design through examples.



(a) Uniform distribution

(b) Weibull distribution

(c) Lognormal distribution

(d) Exponential distribution

Figure 11.3: A few other example probability distributions.

Example 11.2. A robust airfoil optimization.

Consider a simple airfoil optimization, with data from the optimization performed by Nemec et al. [1]. Figure 11.4 shows the drag coefficient of an RAE 2822 airfoil, as a function of Mach number, evaluated by an inviscid compressible flow solver.

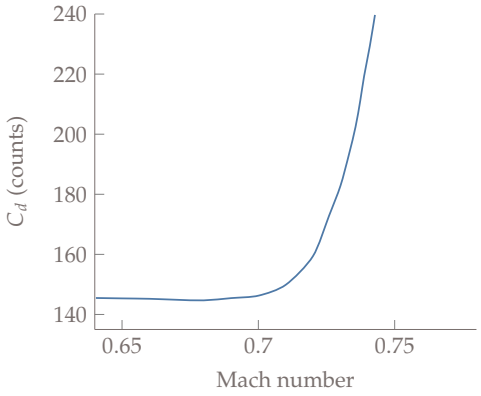


Figure 11.4: Inviscid drag coefficient, in counts, of the RAE 2822 airfoil as a function of Mach number.

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. 11.5 in comparison to the baseline RAE 2822 airfoil.

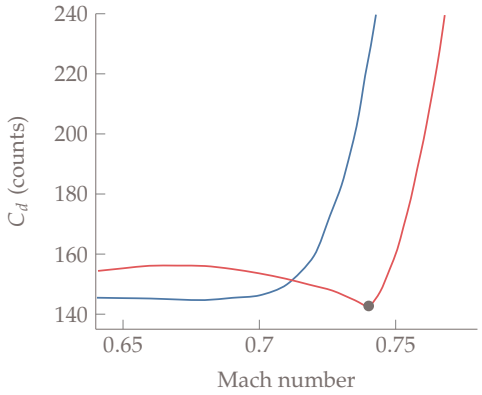


Figure 11.5: The red curve shows the drag of the airfoil optimized to minimize drag at $M = 0.74$, corresponding to the dot. The drag is low at the requested point, but off-design performance is poor.

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 optimization. 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. 11.6.

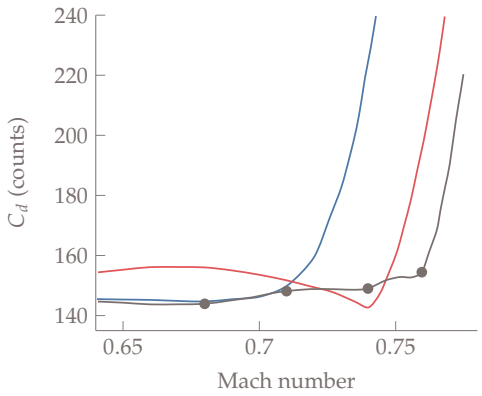


Figure 11.6: The gray curve shows the drag of the airfoil optimized to minimize the average drag at the four denoted points. The robustness of the design is greatly improved.

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 11.3. 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 11.7 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.

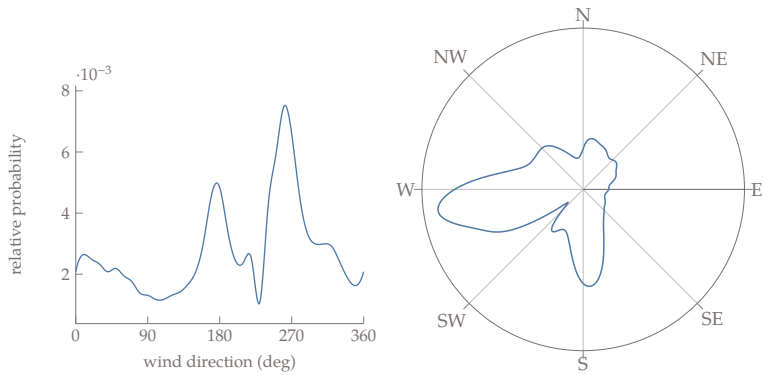


Figure 11.7: Left: probability density function of wind direction. Right: same PDF but visualized as a wind rose.

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°). 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 11.5 explains other ways to perform forward propagation.
Figure 11.8 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 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 deterministic case, which represents over a 12% improvement[a].

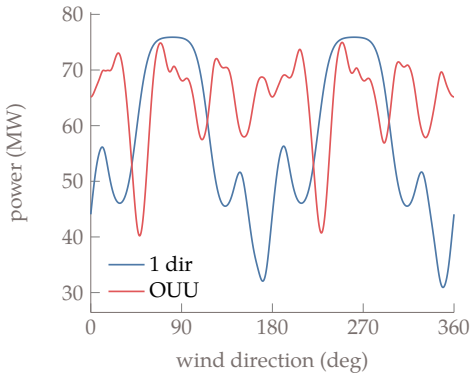


Figure 11.8: Wind farm power, as a function of wind direction, for two cases: optimized deterministically using the most probable direction, optimized under uncertainty.

We can also see the tradeoff in the optimal layouts. The left side of Fig. 11.9 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. 11.9 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.

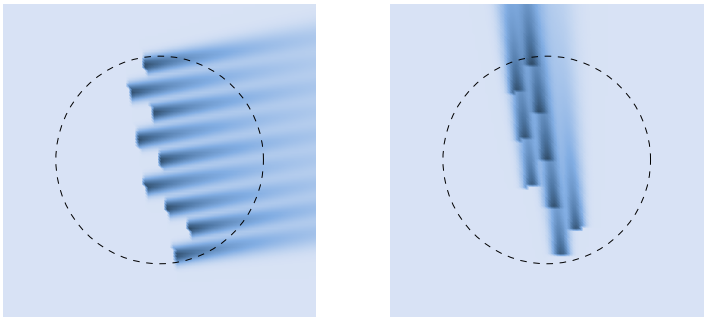


Figure 11.9: Left: Deterministic case with the primary wind direction. Right: Deterministic case with the secondary wind direction.

In contrast, the robust design is shown in Fig. 11.10 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.

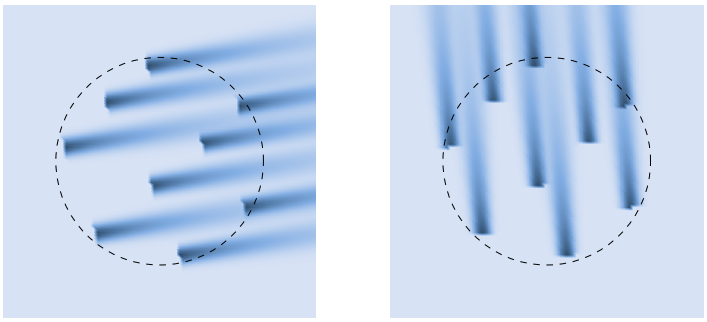


Figure 11.10: Left: OUU case with the primary wind direction. Right: OUU case with the secondary wind direction.

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.

[a]The wind energy community doesn't use expected power directly, but rather annual energy production, which is just the expected power times utilization

Both Examples 11.2 and 11.3 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 7).
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): Prob$[f(x) > f_{crit}]$, which means to minimize the probability that the objective exceeds some critical value.

11.4 Reliability

In addition to affecting the objective function, uncertainty also affects constraints. As with robustness, we begin with an example.

Example 11.4. Reliability with the Barnes function.

Consider the Barnes function shown on the left side of Fig. 11.11. 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).

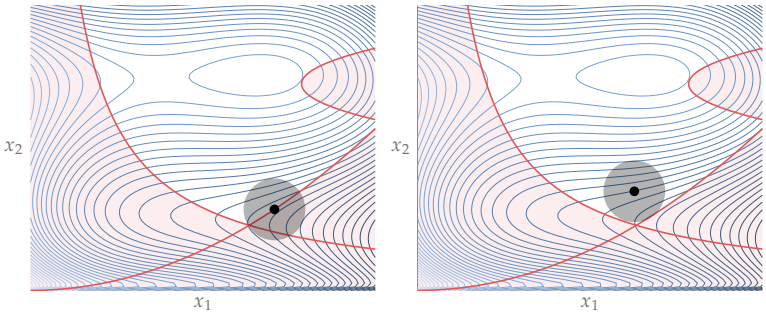


Figure 11.11: Left: The constrained deterministic optimum sits right on the constraint and if there is any variability is likely to violate a constraint. Right: The reliable optimum will still satisfy the constraints even with variability.

Conversely, the right side of Fig. 11.11 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.

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 11.5. Connecting safety factors to reliability.

If we were constraining the stress (σ) in a structure to be less than the material's yield stress (σ_y) we would not want to use a constraint of the form:

$$\sigma(x) \leq \sigma_y \tag{11.11}$$

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.

$$\sigma(x) \leq \eta\sigma_y \tag{11.12}$$

where η is a total safety factor including partial safety factors from loads, materials, failure modes, etc. 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:

$$g(x) \leq 0 \tag{11.13}$$

to the form:

$$\text{Prob}[g(x) \leq 0] \geq R \tag{11.14}$$

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.

11.5 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*[3]. Forward propagation methods propagate input uncertainties through a numerical model to compute output statistics.

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.

[3]Or at least we can make some assumptions that characterize the input uncertainties.

11.5.1 Direct Quadrature

The mean and variance of an output function f are defined as:

$$\mu_f = \int f(x)p(x)dx \tag{11.15}$$

$$\sigma_f^2 = \int f(x)^2 p(x)dx - \mu_f^2 \tag{11.16}$$

and one way to estimate them is to use direct numerical *quadrature*. In other words, the estimation of each integral becomes a summation:

$$\int f(x)dx \approx \sum_i f(x_i)w_i \tag{11.17}$$

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-Konrod, etc.).

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 looks like:

$$\int f(x)dx_1dx_2 \ldots dx_n \approx \sum_i \sum_j \ldots \sum_n f(x_i, x_j, \ldots, x_n)w_i w_j \ldots w_n \tag{11.18}$$

While conceptually straightforward, this approaches faces 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 [2]. 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.

> **Example 11.6.** Sparse grid methods for quadrature.
>
> Figure 11.12 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.

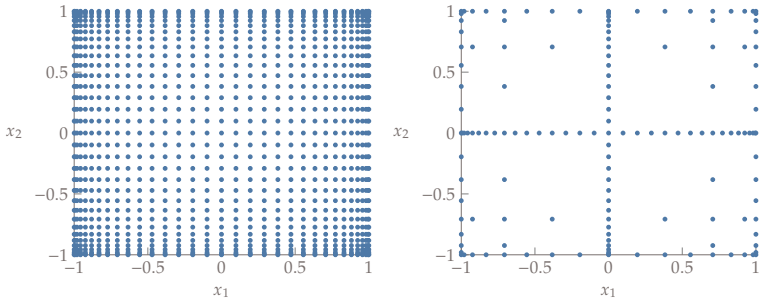


Figure 11.12: Comparison between a two-dimensional full tensor grid (left) and a level 5 sparse grid (right) using the Clenshaw-Curtis exponential rule.

For a problem with dimension d, and approximately N sample points in each dimension, the full tensor grid has a computational complexity of $O(N^d)$, whereas the sparse grid method has a complexity of $O(N(\log N)^{d-1})$ 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.

11.5.2 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. (11.1) and (11.3)).

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: $O(1/\sqrt{N})$. In other words, 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'd 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 Latin Hypercube sampling instead of pure random sampling. Latin Hypercube sampling allows one to better approximate the input distributions with fewer samples, and is discussed further in Section 12.2 on surrogate-based optimization. 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 required, which can be particularly prohibitive if used as part of an OUU problem.

11.5.3 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):

$$\begin{aligned} \mu_f &= f(\mu_x) \\ \sigma_f^2 &= \sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \sigma_{x_i} \right)^2 \end{aligned} \tag{11.19}$$

where x contains all the uncertain variables (design variables and parameters).

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 (Example 11.3).

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 (skewness, kurtosis, etc.).

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. (11.19). 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 methods discussed in Chapter 4 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 [3]. 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 σ_g using Eq. (11.19).
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.

11.5.4 Polynomial Chaos

The final class of forward propagation methods take advantage of the inherit smoothness of the outputs of interest using polynomial approximations. These methods are also known in the literature as spectral expansions. This latter name is more apt because the method is not chaotic, and in fact need not use polynomials. The name polynomial chaos simply came about because the methodology was initially derived for use in a physical theory of chaos [4]. Still, we will use the name polynomial chaos in this section as it is commonly referred to that way.

A general function that depends on uncertain variables x, can be represented as a sum of basis functions ψ_i (usually polynomials) with weights α_i:

$$f(x) = \sum_{i=0}^{\infty} \alpha_i \psi_i(x) \tag{11.20}$$

although in practice we truncate the series after $n + 1$ terms.

$$f(x) \approx \sum_{i=0}^{n} \alpha_i \psi_i(x) \tag{11.21}$$

The required number of terms for a given input dimension d, and polynomial order p is:

$$n + 1 = \frac{(d + p)!}{d!p!} \tag{11.22}$$

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

$$\vec{u} \cdot \vec{v} = 0 \tag{11.23}$$

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:

$$\int_a^b f(x)g(x)dx = 0 \tag{11.24}$$

For our purposes, the weighted inner product is more relevant:

$$\langle f, g \rangle = \int_a^b f(x)g(x)p(x)dx = 0 \tag{11.25}$$

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 doesn't 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 doesn't 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

$$\langle \psi_i, \psi_j \rangle = 0 \text{ if } i \neq j \tag{11.26}$$

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. (11.6) the mean of the function f is given by:

$$\mu_f = \int f(x)p(x)dx \tag{11.27}$$

where $p(x)$ is the PDF for x. Using our polynomial approximation we can express this as:

$$\mu_f = \int \sum_i \alpha_i \psi_i(x)p(x)dx \tag{11.28}$$

The coefficients α_i are constants and so can be taken out of the integral:

$$\mu_f = \sum_i \alpha_i \int \psi_i(x)p(x)dx \tag{11.29}$$

$$= \alpha_0 \int \psi_0(x)p(x)dx + \alpha_1 \int \psi_1(x)p(x)dx + \alpha_2 \int \psi_2(x)p(x)dx + \dots \tag{11.30}$$

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:

$$\mu_f = \alpha_0 \int p(x)dx + \alpha_1\langle\psi_0, \psi_1\rangle + \alpha_2\langle\psi_0, \psi_2\rangle + \dots \tag{11.31}$$

Because the polynomials are orthogonal, all of the terms except the first are zero (see Eq. (11.26)), and by definition of a PDF, we know that the integral of $p(x)$ over the domain must be one (see Eq. (11.5)). Thus, we have the simple result that the mean of the function is simply given by the zeroth coefficient:

$$\mu_f = \alpha_0 \tag{11.32}$$

Using a similar approach we can derive a formula for the variance. By definition, the variance is (Eq. (11.8)):

$$\sigma_f^2 = \int f(x)^2 p(x)dx - \mu_x^2 \tag{11.33}$$

Substituting our polynomial representation and using the same techniques used in deriving the mean:

$$\sigma_f^2 = \int \left(\sum_i \alpha_i \psi_i(x)\right)^2 p(x)dx - \alpha_0^2 \tag{11.34}$$

$$= \sum_i \alpha_i^2 \int \psi_i(x)^2 p(x)dx - \alpha_0^2 \tag{11.35}$$

$$= \alpha_0^2 \int \psi_0^2 p(x)dx + \sum_{i=1}^{n} \alpha_i^2 \int \psi_i(x)^2 p(x)dx - \alpha_0^2 \tag{11.36}$$

$$= \alpha_0^2 + \sum_{i=1}^{n} \alpha_i^2 \int \psi_i(x)^2 p(x)dx - \alpha_0^2 \tag{11.37}$$

$$= \sum_{i=1}^{n} \alpha_i^2 \int \psi_i(x)^2 p(x)dx \tag{11.38}$$

$$= \sum_{i=1}^{n} \alpha_i^2 \langle\psi_i^2\rangle \tag{11.39}$$

$$\tag{11.40}$$

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:

$$\mu_f = \alpha_0 \tag{11.41}$$

$$\sigma_f^2 = \sum_{i=1}^{n} \alpha_i^2 \langle \Psi_i^2 \rangle \tag{11.42}$$

except that Ψ_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. (11.25) 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[4]. Table 11.1 summarizes the polynomials that correspond to some common probability distributions [5]. Referring again to Eq. (11.25), 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 [5].

Table 11.1: 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	$[0, \infty]$

[4] Actually other polynomials can be used, but these particular choices are optimal as they produce exponential convergence.

Example 11.7. Legendre polynomials.

The first few Legendre polynomials are:

$$\begin{aligned}\psi_0 &= 1\\ \psi_1 &= x\\ \psi_2 &= \frac{1}{2}(3x^2 - 1)\\ \psi_3 &= \frac{1}{2}(5x^3 - 3x)\\ &\vdots\end{aligned} \tag{11.43}$$

These polynomials are plotted in Fig. 11.13, and are orthogonal with respect to a uniform probability distribution.

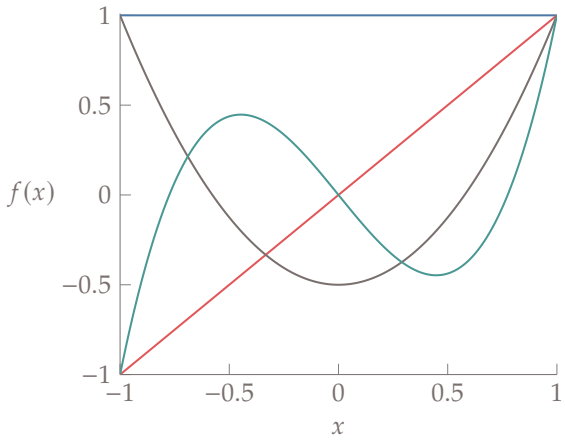


Figure 11.13: The first few Legendre polynomials.

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:

$$\Psi_0(x) = \psi_0(x_1)\psi_0(x_2) = 1 \tag{11.44}$$
$$\Psi_1(x) = \psi_1(x_1)\psi_0(x_2) = x_1 \tag{11.45}$$
$$\Psi_2(x) = \psi_0(x_1)\psi_1(x_2) = x_2 \tag{11.46}$$
$$\Psi_3(x) = \psi_1(x_1)\psi_1(x_2) = x_1 x_2 \tag{11.47}$$
$$\Psi_4(x) = \psi_2(x_1)\psi_0(x_2) = \frac{1}{2}(3x_1^2 - 1) \tag{11.48}$$
$$\Psi_5(x) = \psi_0(x_1)\psi_2(x_2) = \frac{1}{2}(3x_2^2 - 1) \tag{11.49}$$

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. (11.22), with $d = 2, p = 2$, suggests that we should have six basis functions.

Determine Coefficients

With the polynomial basis ψ_i fixed, we need to determine the appropriate coefficients α_i in Eq. (11.21). 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:

$$f(x) = \sum_i \alpha_i \psi_i(x) \tag{11.50}$$

we take the inner product of both sides with respect to ψ_j.

$$\langle f(x), \psi_j \rangle = \sum_i \alpha_i \langle \psi_i, \psi_j \rangle \tag{11.51}$$

Making use of the orthogonality of the basis functions (Eq. (11.26)) we see that all of the terms in the summation are zero except:

$$\langle f(x), \psi_i \rangle = \alpha_i \langle \psi_i^2 \rangle \tag{11.52}$$

or

$$\alpha_i = \frac{1}{\langle \psi_i^2 \rangle} \int f(x)\psi_i(x)p(x)dx \tag{11.53}$$

Note that, as expected, the zeroth coefficient is simply the definition of the mean.

$$\alpha_0 = \int f(x)p(x)dx \tag{11.54}$$

The remaining coefficients must be obtained through multidimensional quadrature using the same types of approaches as discussed in Section 11.5.1 (or even

Section 11.5.2). 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. (11.21)) we need to know $f(x)$ (Eq. (11.53)). 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. (11.21) 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^{th} sample, resulting in a linear system:

$$\begin{bmatrix} \psi_0(x^{(1)}) & \dots & \psi_n(x^{(1)}) \\ \vdots & & \vdots \\ \psi_0(x^{(m)}) & \dots & \psi_n(x^{(m)}) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} f(x^{(1)}) \\ \vdots \\ f(x^{(m)}) \end{bmatrix} \tag{11.55}$$

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.

11.5.5 Summary

Four classes of forward propagation were discussed in this chapter and are summarized below. 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.

Direct quadrature 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 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 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 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.

11.6 Further Notes

- A more general expression for first-order perturbation methods, using the covariance matrix, is shown in [6]. Higher-order Taylor's series can also be used [7], but they are less common because of their increased complexity.
- Several packages exist to facilitate use of polynomial chaos methods [8, 9].

Bibliography

[1] Marian Nemec, David W. Zingg, and Thomas H. Pulliam. Multipoint and multi-objective aerodynamic shape optimization. *AIAA Journal*, 42(6):1057–1065, June 2004.

[2] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 148(5):1042–1045, 1963.

[3] A. Parkinson, C. Sorensen, and N. Pourhassan. A general approach for robust optimal design. *Journal of Mechanical Design*, 115(1):74, 1993. doi:10.1115/1.2919328.

[4] Norbert Wiener. The homogeneous chaos. *American Journal of Mathematics*, 60(4):897, oct 1938. doi:10.2307/2371268.

[5] Michael Eldred, Clayton Webster, and Paul Constantine. Evaluation of non-intrusive approaches for wiener-askey generalized polynomial chaos. In *49th AIAA Structures, Structural Dynamics, and Materials Conference*. American Institute of Aeronautics and Astronautics, apr 2008. doi:10.2514/6.2008-1892.

[6] Ralph C. Smith. *Uncertainty Quantification: Theory, Implementation, and Applications*. SIAM, Dec 2013. ISBN 1611973228.

[7] Dan Cacuci. *Sensitivity & Uncertainty Analysis, Volume 1*. Chapman and Hall/CRC, may 2003. doi:10.1201/9780203498798.

[8] B.M. Adams, L.E. Bauman, W.J. Bohnhoff, K.R. Dalbey, M.S. Ebeida, J.P. Eddy, M.S. Eldred, P.D. Hough, K.T. Hu, J.D. Jakeman, J.A. Stephens, L.P. Swiler, D.M. Vigil, and T.M Wildey. 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, Nov 2015.

[9] Jonathan Feinberg and Hans Petter Langtangen. Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal of Computational Science*, 11:46–57, nov 2015. doi:10.1016/j.jocs.2015.08.008.

CHAPTER 12

Surrogate-Based Optimization

A surrogate model, also known as a response surface model or metamodel, is an approximate model of functional output that acts like a "curve fit" to some underlying data. The goal is to create a surrogate that is much faster (and perhaps smoother) than the original function, but that still retains sufficient accuracy away from known data points.

12.1 Overview

There are various scenarios for which a surrogate model might be useful:

- When the simulation is expensive to evaluate.
- When the simulation is noisy.
- When the model is derived from experimental data (which may be both expensive and noisy).
- When one wants to better understand functional relationships between the inputs and outputs.
- When multiple fidelities are involved (e.g., Euler and Navier-Stokes, or experiments and simulations) 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 are shown in Fig. 12.1. First, sampling methods are used to choose the initial points to evaluate the function, or conduct experiments at. 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 decide on new points to sample and reconstruct the surrogate (infill). This process is repeated until some convergence criteria or maximum number of iterations is reached. The optimization step is no different than those we have already discussed. The new steps we discuss in this chapter are sampling, constructing a surrogate, and infill. Many of the concepts discussed in this chapter are of broader usefulness in optimization beyond just SBO.

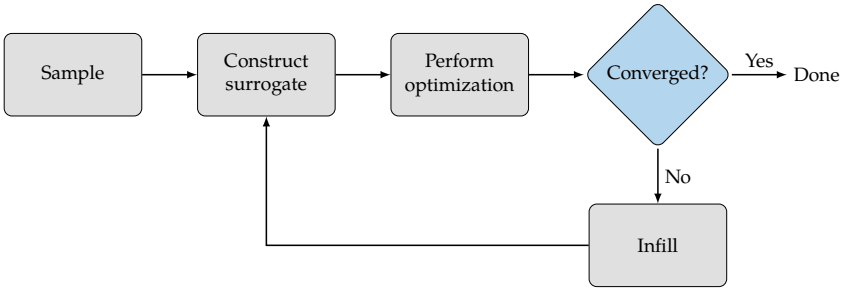


Figure 12.1: Overview of surrogate-based optimization procedure.

12.2 Sampling

Sampling methods select the evaluation points for constructing the initial surrogate. These evaluation points must be chosen carefully.

Example 12.1. 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!

Example 12.1 highlights one of the major challenges of sampling methods: the curse of dimensionality. For SBO, using a large number of variables 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 a random process, but it is much more effective 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. Let's say we wanted only eight samples, we could divide the design space into eight intervals in each dimension as shown in Fig. 12.2.

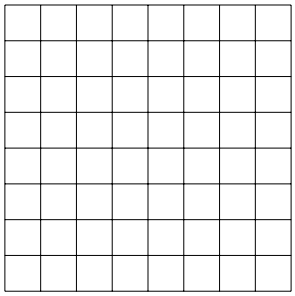
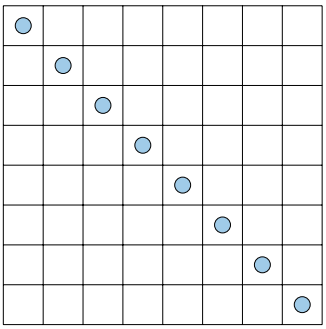


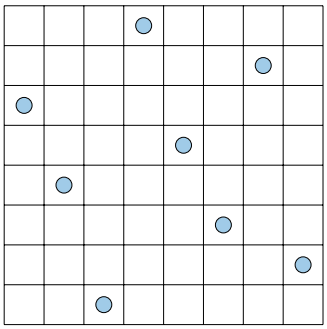
Figure 12.2: A two dimensional design spaces divided into 8 intervals in each dimension.

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 (the popular puzzle game Sudoku is an example of a Latin square). 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. 12.3a. This plan achieves our criteria but clearly doesn't fill the space and likely won't capture

the relationships between design parameters very well. Alternatively Fig. 12.3b has a sample in each row and column while also spanning the space much more effectively.



(a) A sampling strategy whose projection uniformly spans each dimension but doesn't fill the space very well.



(b) A sampling strategy whose projection uniformly spans each dimension and fills the space more effectively.

Figure 12.3: Contrasting sampling strategies that both fulfill the uniform projection requirement.

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 axes 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, normal, etc.), but you can also easily sample from an arbitrary distribution using a technique 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. 12.4 for a normal distribution. Note that this same procedure allows you to use Latin Hypercube sampling with any distribution, simply by generating the samples on a uniform

distribution.

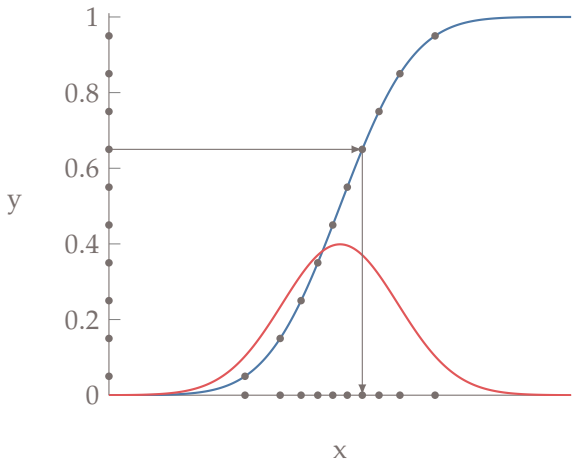


Figure 12.4: An example of inversion sampling with a normal distribution. A few uniform samples are shown on the y-axis. The points are evaluated by the inverse CDF, depicted by the arrows passing through the CDF for a normal distribution (blue curve). If enough samples are drawn, the resulting distribution will be the PDF of a normal distribution (red curve).

In addition to use in SBO, Latin Hypercube Sampling is also very useful in other applications discussed in this book including: initializing a Genetic Algorithm (Section 6.4), initializing a Particle Swarm Method (Section 6.5), performing restarts in a gradient-based method for exploration of multiple minima (Tip 3.2), or choosing the points to run in a Monte Carlo simulation (Section 11.5.2).

12.3 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 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 there will illustrate many of the important concepts in surrogate models.

A general linear regression model looks like:

$$\hat{f} = w^T \psi \tag{12.1}$$

where w is a vector of weights and ψ is a vector of basis functions. Another popular regression model is Kriging, where the basis functions are:

$$\psi^{(i)} = \exp\left(-\sum_j \theta_j |x_j^{(i)} - x_j|^{p_j} \right) \tag{12.2}$$

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 12.2. 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:

$$\psi \in \{1, x_1, x_2, x_3, x_1x_2, x_1x_3, x_2^2, x_1^2x_3 \ldots\} \tag{12.3}$$

To construct a linear regression model there are two tasks: 1) determine what terms to include in ψ, and 2) estimate w to minimize some error. For a given set of ψ 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[1]:

$$\text{minimize} \sum_i \left(\hat{f}(x^{(i)}) - f^{(i)} \right)^2 \tag{12.4}$$

The solution to this optimization problem is called a least squares solution as discussed in Section 9.2.1. Let's rewrite $\mathbf{f}$ in matrix form so that it is compact for all $x^{(i)}$:

$$\mathbf{f} = \Psi w \tag{12.5}$$

[1] 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.

where Ψ is matrix:

$$\Psi = \begin{bmatrix} — & \psi(x^{(1)})^T & — \\ — & \psi(x^{(2)})^T & — \\ & \vdots & \\ — & \psi(x^{(n)})^T & — \end{bmatrix} \tag{12.6}$$

Thus, the same minimization problem can be expressed as:

$$\text{minimize } ||\Psi w - f||^2 \tag{12.7}$$

where

$$f = \begin{bmatrix} f^{(1)} \\ f^{(2)} \\ \vdots \\ f^{(n)} \end{bmatrix} \tag{12.8}$$

The matrix Ψ is of size (m x 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 $||Ax - y||^2$.

> **Practical Tip 12.1.** Least squares is not the same as a linear system solve.
>
> In Julia or Matlab you can solve this with `x = A\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 solve as it would for a full rank n x 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 ψ. 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 12.3. The dangers of overfitting.

Consider a simple underlying function with Gaussian noise added to simulate experimental or noisy computational data.

$$y = 0.2x^4 + 0.2x^3 - 0.9x^2 + \mathcal{N}(0, \sigma), \text{ where } \sigma = 0.75 \tag{12.9}$$

We will create the data at 20 points between $-2 \ldots 2$ (see Fig. 12.5).

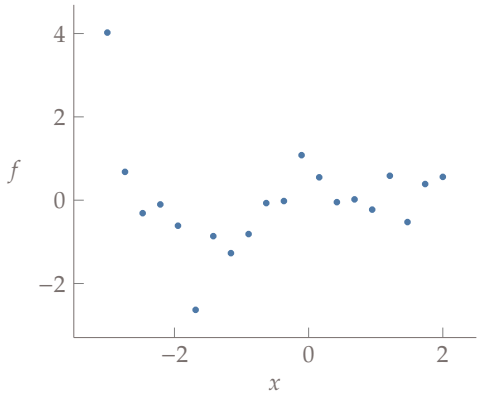


Figure 12.5: Data from a numerical model or experiments.

For a real problem we don't know the underlying function and the dimensionality is often too high to visualize. Determining the right basis to 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. 12.6).

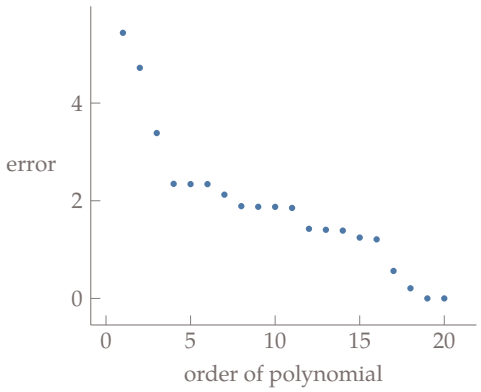


Figure 12.6: Error in fitting data with different order polynomials.

It seems as if the higher the order of the polynomial, then the lower the error. For example, a 10th order polynomial reduces the error to almost zero. The problem of course, 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. 12.7 shows a 19th order fit to the data. The model passes right through the points, but it predictive capability is poor.

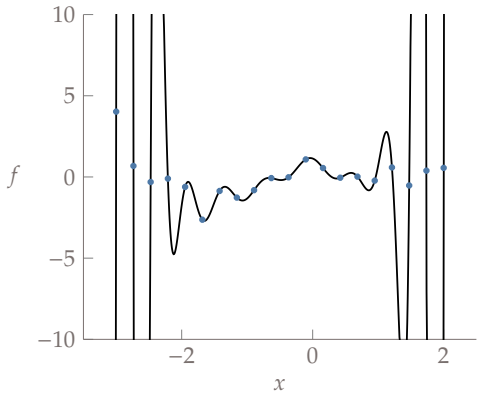


Figure 12.7: A 19th order polynomial fit to the data. Low error, but poor predictive ability.

This is called *overfitting*. Of course, we also want to avoid the opposite problem of underfitting where we don't 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 Example 12.3 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 models (the different options for ψ) 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 models 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 12.4. Cross validation is used to avoid overfitting.

This example continues from Example 12.3. First, we perform k-fold cross-validation using ten divisions. The average error across the divisions using the training data is shown in Fig. 12.8.

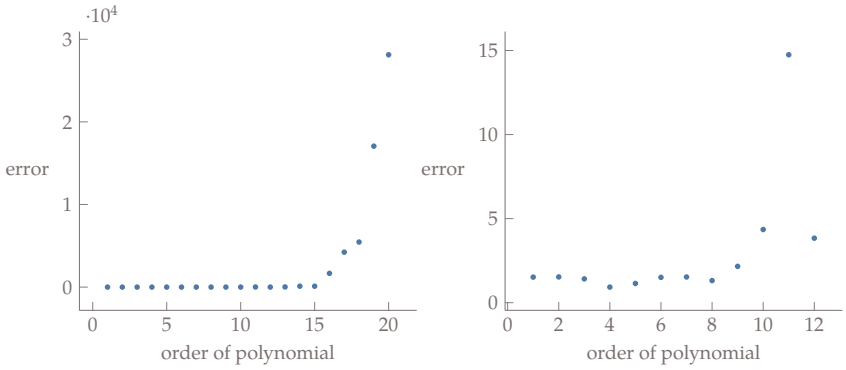


Figure 12.8: Error from k-fold cross validation.

Note that 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 error. 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. 12.9. This model has much better predictive capability.

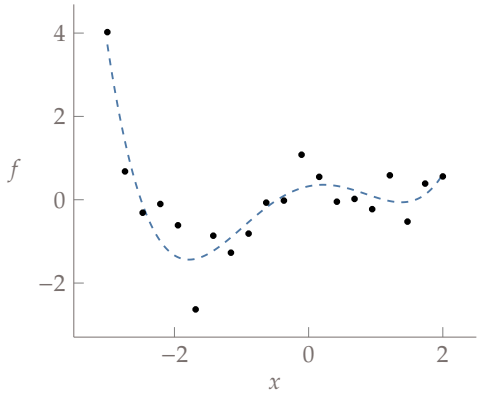


Figure 12.9: A 4th order polynomial fit to the data.

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:

$$\psi^{(i)} = \psi(||x - c^{(i)})||, \tag{12.10}$$

of which one of the more popular models is Kriging:

$$\psi^{(i)} = \exp\left(-\sum_j \theta_j |x_j^{(i)} - x_j|^{p_j}\right) \tag{12.11}$$

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. 12.10. Notice how the uncertainty goes to zero at the known data points, and becomes largest when far from known data points.

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.

12.4 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.

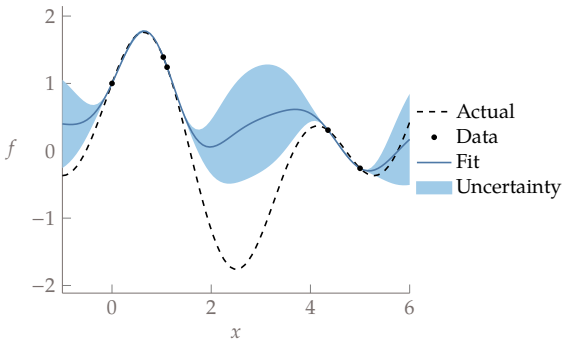


Figure 12.10: A Kriging fit to the input data (circles) and a confidence interval in gray.

For the polynomial models discussed in the previous section the only real option is exploitation. An exploitation-based infill strategy means that we will add an infill point at wherever the surrogate predicts the optimum is located at. The reasoning behind this method, is that in SBO we don't 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 won't 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 don't 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've 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:

$$I(x) = \max(f(x^*) - f(x), 0) \tag{12.12}$$

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, *expected improvement* is simply the

expected value (or mean) of the improvement

$$EI(x) = \mathbb{E}\left[\max(f(x^*) - f(x), 0)\right] \tag{12.13}$$

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 12.5. Expected improvement.

Consider a 1D function with data points and a fit as shown in Fig. 12.11. This data is based on an example from Rajnarayan et al. [1]. The best point we've 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.

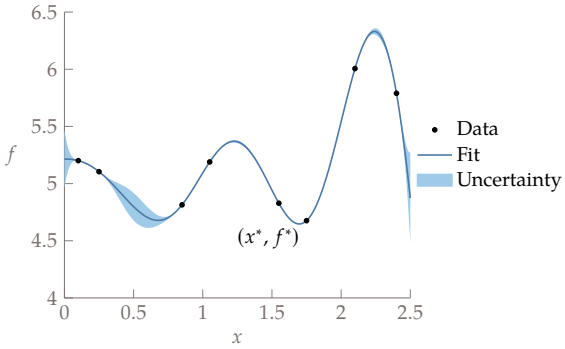


Figure 12.11: A one-dimensional function with a Gaussian process model surrogate fit and uncertainty.

Now imagine we want to evaluate this function at some new test point $x_{test} = 0.5$. In Fig. 12.12 the probability distribution for the objective at x_{test} is shown in blue (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.

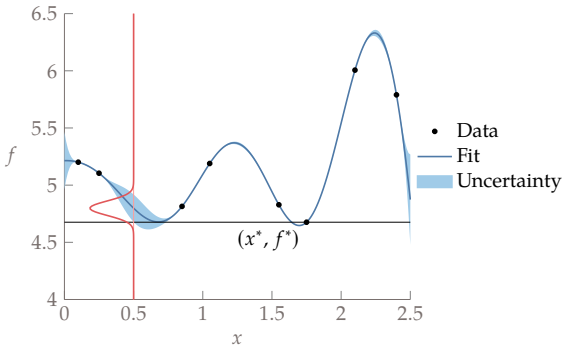


Figure 12.12: At a given test point ($x_{test} = 0.5$) we highlight the probability distribution and the expected improvement in the shaded blue region.

Now, let's evaluate the expected improvement not just at $x_{test} = 0.5$ but across the domain. The result is shown by the green function. 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 doesn't 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.

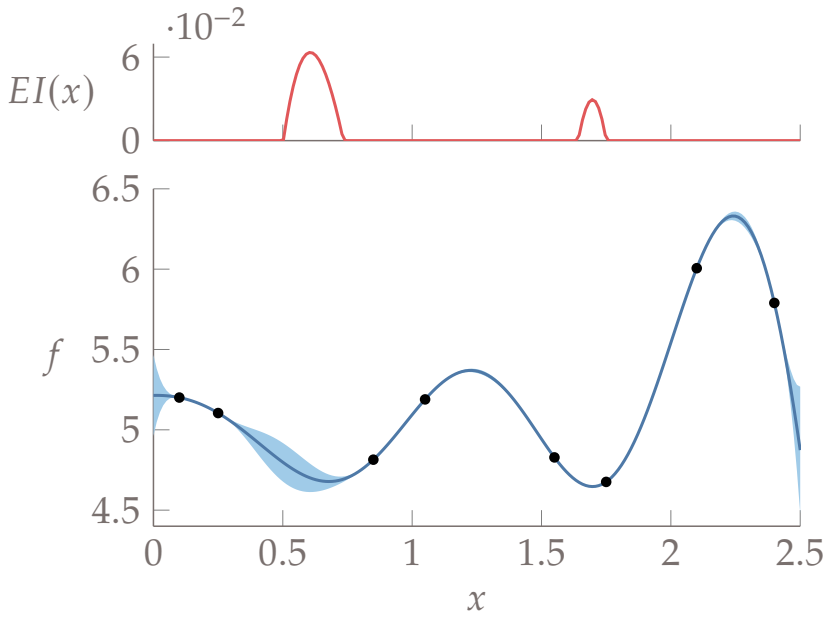


Figure 12.13: The process is repeated by evaluating expected improvement across the domain.

12.5 Further Notes

- The surrogate modeling toolbox is a useful package for surrogate modeling with a particular focus on providing derivatives for use in gradient-based optimization.
- Engineering Design via Surrogate Modelling [2] provides a nice introduction to this topic with much more depth than can be provided in this chapter.

Bibliography

[1] Dev Rajnarayan, Alex Haas, and Ilan Kroo. A multifidelity gradient-free optimization method and application to aerodynamic design. *12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Sep 2008. doi:10.2514/6.2008-6020. URL http://dx.doi.org/10.2514/6.2008-6020.

[2] Alexander Forrester, András Sobester, and Andy Keane. *Engineering Design via Surrogate Modelling: A Practical Guide*. John Wiley & Sons, Sep 2008. ISBN 0470770791.

Exercises

1. *Latin hypercube sampling*. Use a LHS package to create and plot eight points across two dimensions with uniform projection in both dimensions (essentially recreating Fig. 12.3b but with a different layout because of randomness).

2. *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). Is the projection of points consistent with a normal distribution in each dimension?

3. *Linear regression*. Create some sample training data by simulating a function $f(x) = 3x^2 + 2x + 1$ at 20 points from $x = [-2, 2]$ and add random Gaussian noise at each x with mean 0 and standard deviation 1. Pretend this is given training data where the underlying function is unknown. Use linear regression to determine the coefficients for a polynomial basis of $[x^2, x, 1]$. The answer should be close to the function we constructed $w = [3, 2, 1]$ (it won't be that exactly since we added noise to the function). Plot your fit against the training data.

4. *Cross validation*. Create the noisy function shown in Example 12.3. Show that the error in fitting data decreases with higher order polynomials (Fig. 12.6). Visualize one of the high-order fits (Fig. 12.7). Now use cross-validation (the example uses k-fold cross validation with ten divisions) to recreate similar plots to (Fig. 12.8) and finally plot one of the fits with low error against the testing set (Fig. 12.9).

APPENDIX A

Mathematics Review

This chapter briefly reviews some select mathematical concepts that are used throughout the book.

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:

$$\frac{d}{dx}\left(f(g(x))\right) = \frac{df}{dg}\frac{dg}{dx} \tag{A.1}$$

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:

$$\frac{d}{dx}\left(f(g(x))\right) = \frac{d}{dg}(\sin(g))\frac{d}{dx}(x^2) = \cos(x^2)(2x) \tag{A.2}$$

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:

$$\frac{d}{dx}\left(f(g(x),h(x))\right) = \frac{\partial f}{\partial g}\frac{dg}{dx} + \frac{\partial f}{\partial h}\frac{dh}{dx} \tag{A.3}$$

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,

$$\frac{d}{dx}(f(g_1(x),\ldots,g_n(x))) = \sum_{i=1}^{n}\left(\frac{\partial f}{\partial g_i}\frac{dg_i}{dx}\right) \tag{A.4}$$

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:

$$\frac{\partial f}{\partial x} = 2x \tag{A.5}$$

whereas the total derivative of f with respect to x is:

$$\begin{aligned}\frac{df}{dx} &= \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx}\\ &= 2x + 2y\cos(x)\\ &= 2x + 2\sin(x)\cos(x)\end{aligned} \tag{A.6}$$

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:

$$f(x) = x^2 + \sin^2(x) \tag{A.7}$$

$$\frac{df}{dx} = 2x + 2\sin(x)\cos(x) \tag{A.8}$$

Example A.3. Multivariable chain rule.

Expanding on our single variable example, let $g(x) = \cos(x)$ and $h(x) = \sin(x)$ and $f(g, h) = g^2h^3$. Then $f(g(x), h(x)) = \cos^2(x)\sin^3(x)$ Applying

Eq. (A.3) we have:

$$\begin{aligned}\frac{d}{dx}(f(g(x),h(x))) &= \frac{\partial f}{\partial g}\frac{dg}{dx} + \frac{\partial f}{\partial h}\frac{dh}{dx} \\ &= 2gh^3\frac{dg}{dx} + g^2 3h^2\frac{dh}{dx} \\ &= -2gh^3\sin(x) + g^2 3h^2\cos(x) \\ &= -2\cos(x)\sin^4(x) + 3\cos^3(x)\sin^2(x)\end{aligned} \tag{A.9}$$

A.2 Vector and Matrix Norms

The most familiar norm is the Euclidean norm, which represents the Euclidean distance between two points:

$$\|x\|_2 = (x_1^2 + x_2^2 + \ldots x_n^2)^{1/2} \tag{A.10}$$

This norm is also called a 2-norm. More generally, we can refer to a class of norms called p-norms:

$$\|x\|_p = (|x_1|^p + |x_2|^p + \ldots |x_n|^p)^{1/p} \tag{A.11}$$

Of all the p-norms, there are three that are most commonly used. One is the 2-norm, which we are already familiar with. The others are the 1-norm and the ∞-norm. From the above definition, we see that the 1-norm is the sum of the absolute values of all the entries in x.

$$\|x\|_1 = |x_1| + |x_2| + \ldots |x_n| \tag{A.12}$$

The application of ∞ in the p-norm definition is perhaps less obvious, but as $p \to \infty$ 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:

$$\|x\|_\infty = \max_i |x_i| \tag{A.13}$$

The infinity norm is commonly used in connection with optimization convergence criteria.

Several norms for matrices exist as well. In this book we use the Frobenius norm:

$$\|H\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} H_{ij}^2}, \tag{A.14}$$

where H is an $m \times n$ matrix.

A.3 Matrix Multiplication

In this book we always assume that a vector u is a column vector, and thus u^T is a row vector. The notation $u \in \mathbb{R}^n$ means u is a vector of real numbers with length n whereas $A \in \mathbb{R}^{mxn}$ means A is a matrix of real numbers with m rows and n columns.

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:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{A.15}$$

where $C \in \mathbb{R}^{mxp}$. Notice that two matrices can be multiplied only if there inner dimensions are equal (n in this case). The remaining products discussed in section are just special cases of of matrix multiplication, but are common enough that we discuss them separately.

A.3.1 Vector-Vector Products

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:

$$u^T v = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^{n} u_i v_i \tag{A.16}$$

Notice that the order doesn't matter:

$$u^T v = v^T u \tag{A.17}$$

In Euclidean space, where vectors possess magnitude and direction, the inner product is defined as

$$u^T v = ||u||\,||v|| \cos(\theta) \tag{A.18}$$

where $||\cdot||$ indicates a 2-norm, and θ 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.

$$uv^T = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_n \\ u_2v_1 & u_2v_2 & \cdots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \cdots & u_mv_n \end{bmatrix} \tag{A.19}$$

or in index form:

$$(uv^T)_{ij} = u_iv_j \tag{A.20}$$

A.3.2 Matrix-Vector Products

Consider multiplying a matrix $A \in \mathbb{R}^{mxn}$ by a vector $u \in \mathbb{R}^n$. The result is a vector $v \in \mathbb{R}^m$.

$$v = Au \Rightarrow v_i = \sum_{j=1}^{n} A_{ij}u_j \tag{A.21}$$

We can see that the entries in v are dot products between the rows of A and u:

$$v = \begin{bmatrix} \text{—} & a_1^T & \text{—} \\ \text{—} & a_2^T & \text{—} \\ & \vdots & \\ \text{—} & a_m^T & \text{—} \end{bmatrix} u \tag{A.22}$$

where a_j^T is the j^{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:

$$v = \begin{bmatrix} | \\ a_1 \\ | \end{bmatrix} u_1 + \begin{bmatrix} | \\ a_2 \\ | \end{bmatrix} u_2 + \ldots + \begin{bmatrix} | \\ a_n \\ | \end{bmatrix} u_n \tag{A.23}$$

where a_i are the columns of A.

We can also multiply by a vector on the left, instead of on the right:

$$v^T = u^TA \tag{A.24}$$

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:

$$\alpha = u^T A u = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \tag{A.25}$$

or in index form:

$$\alpha = \sum_{i=1}^{n} \sum_{j=1}^{n} u_i A_{ij} u_j \tag{A.26}$$

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:

$$A_{ij} = 0 \text{ for all } i \neq j \tag{A.27}$$

The *identity matrix* I is a special diagonal matrix where all diagonal components are one.

The *transpose* of a matrix is defined as:

$$[A^T]_{ij} = A_{ji} \tag{A.28}$$

Note that:

$$(A^T)^T = A \tag{A.29}$$

$$(A + B)^T = A^T + B^T \tag{A.30}$$

$$(AB)^T = B^T A^T \tag{A.31}$$

A *symmetric matrix* is one where the matrix is equal to its transpose:

$$A_{ij} = A_{ji} \tag{A.32}$$

The *inverse* of a matrix satisfies:

$$AA^{-1} = I = A^{-1}A \tag{A.33}$$

Not all matrices are invertible. Some common properties for inverses are:

$$(A^{-1})^{-1} = A \tag{A.34}$$
$$(AB)^{-1} = B^{-1}A^{-1} \tag{A.35}$$
$$[A^{-1}]^T = [A^T]^{-1} \tag{A.36}$$

A symmetric matrix A is *positive definite* if for all vectors $\mathbf{x}$ in the real space:

$$x^T Ax > 0 \tag{A.37}$$

Similarly, a *positive semi-definite* matrix satisfies the condition:

$$x^T Mx \geq 0 \tag{A.38}$$

and a *negative definite* matrix satisfies the condition

$$x^T Mx < 0 \tag{A.39}$$

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

$$f(x) = a^T x + b = \sum_{i=1}^{n} a_i x_i + b_i \tag{A.40}$$

where a, x, and b are vectors of length n, and a_i, x_i, and b_i are the ith 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

$$\frac{\partial}{\partial x_k}\left[\sum_{i=1}^{n} a_i x_i + b_i\right] = a_k \tag{A.41}$$

Thus:

$$\nabla_x(a^T x + b) = a \tag{A.42}$$

If we now look at the quadratic form presented in Appendix A.3.3 we have a general quadratic function

$$f(x) = x^T Ax + b^T x + c \tag{A.43}$$

where x, b and c are still a vectors of length n, and A is an n by n symmetric matrix. In index notation, f looks like

$$f(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} x_i a_{ij} x_j \right) + b_i x_i + c_i \tag{A.44}$$

For convenience, we'll separate the diagonal terms from the off diagonal terms leaving us with

$$f(x) = \sum_{i=1}^{n} \left[a_{ii} x_i^2 + b_i x_i + c_i \right] + \sum_{j \neq i} x_i a_{ij} x_j \tag{A.45}$$

Now we take the partial derivatives with respect to x_k as before yielding

$$\frac{\partial f}{\partial x_k} = 2a_{kk} x_k + b_k + \sum_{j \neq i} x_j a_{jk} + \sum_{j \neq i} a_{kj} x_j \tag{A.46}$$

We now move the diagonal terms back into the sums to get

$$\frac{\partial f}{\partial x_k} = b_k + \sum_{j=1}^{n} (x_j a_{jk} + a_{kj} x_j) \tag{A.47}$$

which we can put back into matrix form as:

$$\nabla_x f(x) = A^T x + Ax + b \tag{A.48}$$

If A is symmetric then $A^T = A$ and thus:

$$\nabla_x (x^T A x + b^T x + c) = 2Ax + b \tag{A.49}$$

A.6 Series Expansion

Series expansion is a mathematical method to describe a given function in terms of a series of other, often simpler, functions. There are various types of series, one common series used in many engineering applications is the polynomial expansion referred to as the Taylor series (or sometimes the Maclaurin series when expanding about zero).

The theory behind the Taylor series states that, given a function, one can express that function as a polynomial or a series of monomial functions. For this to be true, we require that the function be analytic, that is, differentiable at and in a neighborhood of the point about which we are applying our series expansion. This will become clearer when we look at how to obtain the Taylor series.

If we start with some arbitrary function, $f(x)$, that we want to express as a general polynomial at an arbitrary point, x_0:

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \ldots + a_n(x - x_0)^n + \ldots \tag{A.50}$$

If we want the first term to be as close as possible then a_0 must be equal to $f(x_0)$ since all the other terms go to zero at $x = x_0$. To find the other coefficients, we systematically take the derivative of both sides and apply the same pattern to the first non-zero term (which will be a constant). For the first derivative we get

$$f'(x) = a_1 + 2a_2(x - x_0) + \ldots + na_n(x - x_0)^{n-1} \tag{A.51}$$

where we see that a_1 must be $f'(x_0)$. The pattern for the coefficients reveals itself to be

$$a_n = \frac{f^{(n)}(x_0)}{n!} \tag{A.52}$$

Plugging these into our original polynomial gives us the Taylor series

$$f(x) = \sum_{n=0}^{\infty} \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) \tag{A.53}$$

For optimization, we often want to find the value of a function at some distance, Δx, from x with the approximation centered at x. Following the same process, we find the Taylor series to be

$$\begin{aligned} f(x + \Delta x) &= \sum_{n=0}^{\infty} \frac{\Delta x^n}{n!} f^{(n)}(x_0) \\ &= f(x_0) + \Delta x f'(x_0) + \frac{\Delta x^2}{2} f''(x_0) + \ldots \end{aligned} \tag{A.54}$$

The series will be truncated at some finite number of terms, for example $n = 2$ for a quadratic approximation.

The Taylor series in multiple dimensions is similar, except that the function derivatives are the multi-dimensional versions, that is, the gradient for the first derivative and the Hessian for the second derivative. Also, we need to define a direction along which we want to approximate, since that information is not inherent as it is in a 1-D function. For a quadratic approximation along a direction vector $\hat{p}$:

$$f(x + \Delta x\hat{p}) = f(x) + \Delta x \nabla f(x)^T \hat{p} + \frac{\Delta x^2}{2} \hat{p}^T H(x) \hat{p} + O\left(\Delta x^3\right) \tag{A.55}$$

Notice, the first derivative is just a directional derivative as we would expect. Typically, we would express this in terms of the vector with both magnitude and direction $s = \Delta x\hat{p}$:

$$f(x + s) = f(x) + \nabla f(x)^T s + \frac{1}{2} s^T H(x) s + O\left(\|s\|^3\right) \tag{A.56}$$