CHAPTER 10

Constrained Optimization

CONTENTS

We continue our consideration of optimization problems by studying the *constrained* case. These problems take the following general form:

$$\begin{aligned} &\text{minimize } f(\vec{x}) \\ &\text{subject to } g(\vec{x}) = \vec{0} \\ &\qquad\qquad h(\vec{x}) \geq \vec{0}. \end{aligned}$$

Here, $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^n \to \mathbb{R}^p$; we call f the *objective function* and the expressions $g(\vec{x}) = \vec{0}, h(\vec{x}) \geq \vec{0}$ the *constraints*.

This form is extremely generic, so algorithms for solving such problems in the absence of additional assumptions on f, g, or h are subject to degeneracies such as local minima and lack of convergence. In fact, this general problem encodes other problems we already have considered. If we take $f(\vec{x}) = h(\vec{x}) \equiv 0$, then this constrained optimization becomes root-finding on g (Chapter 8), while if we take $g(\vec{x}) = h(\vec{x}) \equiv \vec{0}$, it reduces to unconstrained optimization on f (Chapter 9).

Despite this bleak outlook, optimization methods handling the general constrained problem can be valuable even when f, g, and h do not have strong structure. In many cases, especially when f is heuristic anyway, finding a feasible $\vec{x}$ for which $f(\vec{x}) < f(\vec{x}_0)$ starting from an initial guess $\vec{x}_0$ still represents an improvement from the starting point. One application of this philosophy would be an economic system in which f measures costs; since we wish to minimize costs, *any* $\vec{x}$ decreasing f is a useful—and profitable—output.

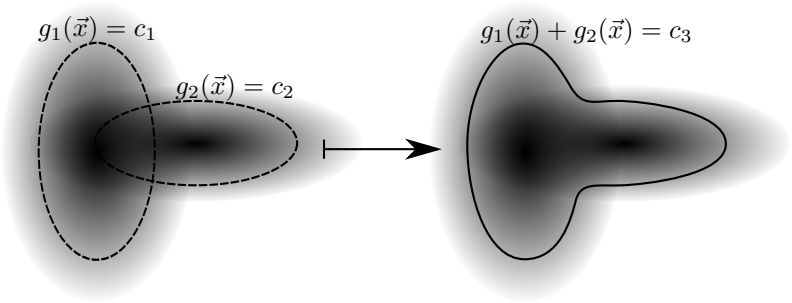


Figure 10.1 "Blobby" shapes are constructed as level sets of a linear combination of functions.

10.1 MOTIVATION

Constrained optimization problems appear in nearly any area of applied math, engineering, and computer science. We already listed many applications of constrained optimization when we discussed eigenvectors and eigenvalues in Chapter 6, since this problem for symmetric matrices $A \in \mathbb{R}^{n \times n}$ can be posed as finding critical points of $\vec{x}^\top A\vec{x}$ subject to $\|\vec{x}\|_2 = 1$. The particular case of eigenvalue computation admits special algorithms that make it a simpler problem. Here, however, we list other optimization problems that do not enjoy the unique structure of eigenvalue problems:

Example 10.1 (Geometric projection)**.** Many shapes S in $\mathbb{R}^n$ can be written *implicitly* in the form $g(\vec{x}) = 0$ for some g. For example, the unit sphere results from taking $g(\vec{x}) \equiv \|\vec{x}\|_2^2 - 1$, while a cube can be constructed by taking $g(\vec{x}) = \|\vec{x}\|_1 - 1$. Some 3D modeling environments allow users to specify "blobby" objects, as in Figure 10.1, as zero-value level sets of $g(\vec{x})$ given by

$$g(\vec{x}) \equiv c + \sum_i a_i e^{-b_i \|\vec{x} - \vec{x}_i\|_2^2}.$$

Suppose we are given a point $\vec{y} \in \mathbb{R}^3$ and wish to find the closest point $\vec{x} \in S$ to $\vec{y}$. This problem is solved by using the following constrained minimization:

$$\begin{aligned} &\text{minimize}_{\vec{x}} \ \|\vec{x} - \vec{y}\|_2 \\ &\text{subject to } g(\vec{x}) = 0. \end{aligned}$$

Example 10.2 (Manufacturing)**.** Suppose you have m different materials; you have s_i units of each material i in stock. You can manufacture k different products; product j gives you profit p_j and uses c_{ij} of material i to make. To maximize profits, you can solve the following optimization for the amount x_j you should manufacture of each item j:

$$\begin{aligned} &\text{maximize}_{\vec{x}} \ \sum_{j=1}^{k} p_j x_j \\ &\text{subject to } x_j \geq 0 \, \forall j \in \{1, \ldots, k\} \\ &\qquad \sum_{j=1}^{k} c_{ij} x_j \leq s_i \, \forall i \in \{1, \ldots, m\}. \end{aligned}$$

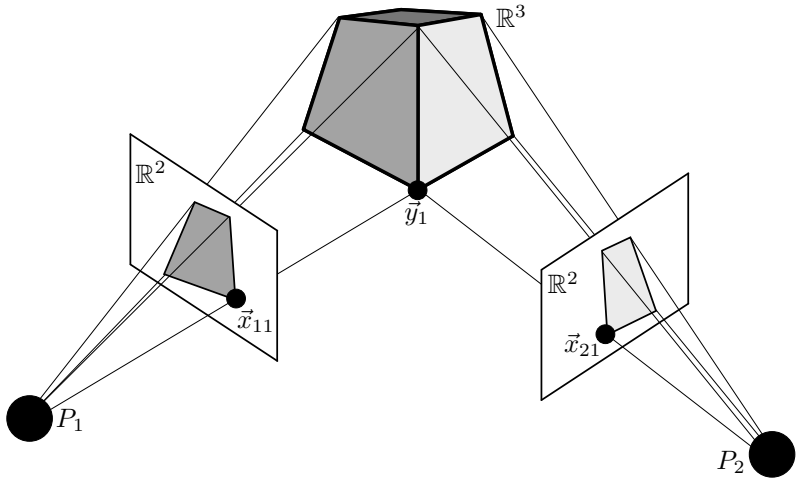


Figure 10.2 Notation for bundle adjustment with two images. Given corresponding points $\vec{x}_{ij}$ marked on images, bundle adjustment simultaneously optimizes for camera parameters encoded in P_i and three-dimensional positions $\vec{y}_j$.

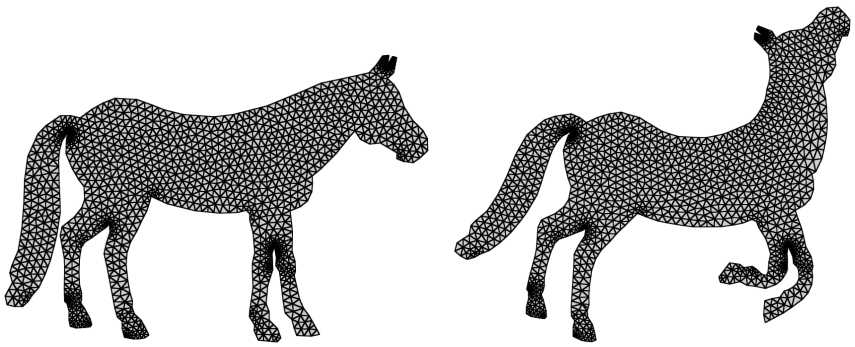
The first constraint ensures that you do not make negative amounts of any product, and the second ensures that you do not use more than your stock of each material. This optimization is an example of a *linear program*, because the objective and constraints are all linear functions. Linear programs allow for inequality constraints, so they cannot always be solved using Gaussian elimination.

Example 10.3 (Nonnegative least-squares)**.** We already have seen numerous examples of least-squares problems, but sometimes negative values in the solution vector might not make sense. For example, in computer graphics, an animated model can be expressed as a deforming bone structure plus a meshed "skin"; for each point on the skin a list of weights can be computed to approximate the influence of the positions of the bone joints on the position of the skin vertices [67]. Such weights should be constrained to be nonnegative to avoid degenerate behavior while the surface deforms. In such a case, we can solve the "nonnegative least-squares" problem:

$$\begin{aligned} &\text{minimize}_{\vec{x}} \ \|A\vec{x} - \vec{b}\|_2 \\ &\text{subject to } x_i \geq 0 \ \forall i. \end{aligned}$$

Some machine learning methods leverage the *sparsity* of nonnegative least-squares solutions, which often lead to optimal vectors $\vec{x}$ with $x_i = 0$ for many indices i [113].

Example 10.4 (Bundle adjustment)**.** In computer vision, suppose we take pictures of an object from several angles. A natural task is to reconstruct the three-dimensional shape of the object from these pictures. To do so, we might mark a corresponding set of points on each image; we can take $\vec{x}_{ij} \in \mathbb{R}^2$ to be the position of feature point j on image i, as in Figure 10.2. In reality, each feature point has a position $\vec{y}_j \in \mathbb{R}^3$ in space, which we would like to compute. Additionally, we must find the positions of the cameras themselves, which we can represent as unknown projection matrices P_i.



(a) Original (b) Deformed

Figure 10.3 As-rigid-as-possible (ARAP) optimization generates the deformed mesh on the right from the original mesh on the left given target positions for a few points on the head, feet, and torso.

The problem of estimating the $\vec{y}_j$'s and P_i's, known as *bundle adjustment*, can be posed as an optimization:

$$\text{minimize}_{\vec{y}_j, P_i} \sum_{ij} \|P_i \vec{y}_j - \vec{x}_{ij}\|_2^2$$
$$\text{such that } P_i \text{ is orthogonal } \forall i.$$

The orthogonality constraint ensures that the camera transformations could have come from a typical lens.

Example 10.5 (As-rigid-as-possible deformation)**.** The "as-rigid-as-possible" (ARAP) modeling technique is used in computer graphics to deform two- and three-dimensional shapes in real time for modeling and animation software [116]. In the planar setting, suppose we are given a two-dimensional triangle mesh, as in Figure 10.3(a). This mesh consists of a collection of vertices V connected into triangles by edges $E \subseteq V \times V$; we will assume each vertex $v \in V$ is associated with a position $\vec{x}_v \in \mathbb{R}^2$. Furthermore, assume the user manually moves a subset of vertices $V_0 \subset V$ to target positions $\vec{y}_v \in \mathbb{R}^2$ for $v \in V_0$ to specify a potential deformation of the shape. The goal of ARAP is to deform the remainder $V \backslash V_0$ of the mesh vertices elastically, as in Figure 10.3(b), yielding a set of new positions $\vec{y}_v \in \mathbb{R}^2$ for each $v \in V$ with $\vec{y}_v$ fixed by the user when $v \in V_0$.

The least-distorting deformation of the mesh is a *rigid* motion, meaning it rotates and translates but does not stretch or shear. In this case, there exists an *orthogonal* matrix $R \in \mathbb{R}^{2\times 2}$ so that the deformation satisfies $\vec{y}_v - \vec{y}_w = R(\vec{x}_v - \vec{x}_w)$ for any edge $(v, w) \in E$. But, if the user wishes to stretch or bend part of the shape, there might not exist a single R rotating the entire mesh to satisfy the position constraints in V_0.

To loosen the single-rotation assumption, ARAP asks that a deformation is *approximately* or *locally* rigid. Specifically, no single vertex on the mesh should experience more than a little stretch or shear, so in a neighborhood of each vertex $v \in V$ there should exist an orthogonal matrix R_v satisfying $\vec{y}_v - \vec{y}_w \approx R_v(\vec{x}_v - \vec{x}_w)$ for any $(v, w) \in E$. Once again applying least-squares, we define the as-rigid-as-possible deformation of the mesh to be

the one mapping $\vec{x}_v \mapsto \vec{y}_v$ for all $v \in V$ by solving the following optimization problem:

$$\begin{aligned} \text{minimize}_{R_v, \vec{y}_v} & \sum_{v \in V} \sum_{(v,w) \in E} \|R_v(\vec{x}_v - \vec{x}_w) - (\vec{y}_v - \vec{y}_w)\|_2^2 \\ \text{subject to } & R_v^\top R_v = I_{2\times 2} \; \forall v \in V \\ & \vec{y}_v \text{ fixed } \forall v \in V_0. \end{aligned}$$

We will suggest one way to solve this optimization problem in Example 12.5.

10.2 THEORY OF CONSTRAINED OPTIMIZATION

In our discussion, we will assume that f, g, and h are differentiable. Some methods exist that only make weak continuity or Lipschitz assumptions, but these techniques are quite specialized and require advanced analytical consideration.

10.2.1 Optimality

Although we have not yet developed algorithms for general constrained optimization, we have made use of the *theory* of these problems. Specifically, recall the method of Lagrange multipliers, introduced in Theorem 1.1. In this technique, critical points of $f(\vec{x})$ subject to $g(\vec{x}) = \vec{0}$ are given by critical points of the *unconstrained* Lagrange multiplier function

$$\Lambda(\vec{x}, \vec{\lambda}) \equiv f(\vec{x}) - \vec{\lambda} \cdot \vec{g}(\vec{x})$$

with respect to both $\vec{\lambda}$ and $\vec{x}$ simultaneously. This theorem allowed us to provide variational interpretations of eigenvalue problems; more generally, it gives an alternative criterion for $\vec{x}$ to be a critical point of an *equality-constrained* optimization.

As we saw in Chapter 8, even finding a feasible $\vec{x}$ satisfying the constraint $g(\vec{x}) = \vec{0}$ can be a considerable challenge even before attempting to minimize $f(\vec{x})$. We can separate these issues by making a few definitions:

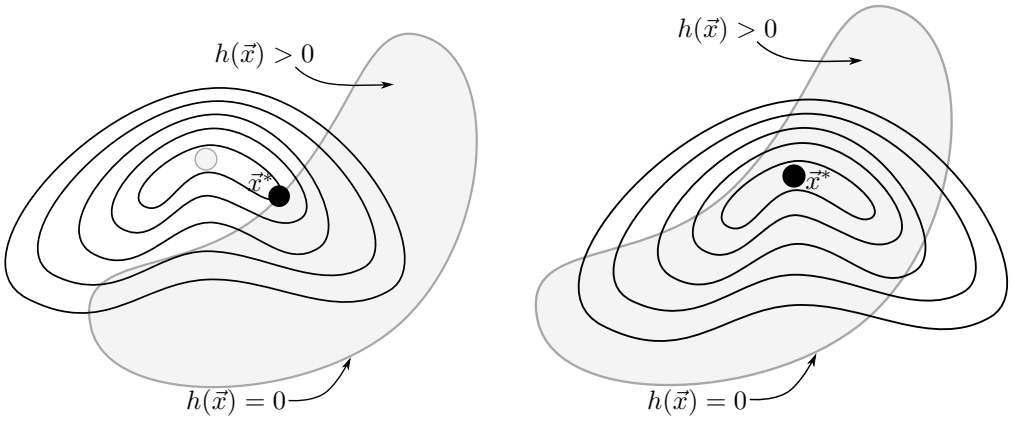
Definition 10.1 (Feasible point and feasible set)**.** A *feasible point* of a constrained optimization problem is any point $\vec{x}$ satisfying $g(\vec{x}) = \vec{0}$ and $h(\vec{x}) \geq \vec{0}$. The *feasible set* is the set of all points $\vec{x}$ satisfying these constraints.

Definition 10.2 (Critical point of constrained optimization)**.** A critical point of a constrained optimization satisfies the constraints and is also a local maximum, minimum, or saddle point of f within the feasible set.

10.2.2 KKT Conditions

Constrained optimizations are difficult because they simultaneously solve root-finding problems (the $g(\vec{x}) = \vec{0}$ constraint), satisfiability problems (the $h(\vec{x}) \geq \vec{0}$ constraint), and minimization (on the function f). As stated in Theorem 1.1, Lagrange multipliers allow us to turn equality-constrained minimization problems into root-finding problems on Λ. To push our differential techniques to complete generality, we must find a way to add inequality constraints $h(\vec{x}) \geq \vec{0}$ to the Lagrange multiplier system.

Suppose we have found a local minimum subject to the constraints, denoted $\vec{x}^*$. For each inequality constraint $h_i(\vec{x}^*) \geq 0$, we have two options:



Active constraint
$h(\vec{x}^*) = 0$

Inactive constraint
$h(\vec{x}^*) > 0$

Figure 10.4 Active and inactive constraints $h(\vec{x}) \geq 0$ for minimizing a function whose level sets are shown in black; the region $h(\vec{x}) \geq 0$ is shown in gray. When the $h(\vec{x}) \geq 0$ constraint is active, the optimal point $\vec{x}^*$ is on the border of the feasible domain and would move if the constraint were removed. When the constraint is inactive, $\vec{x}^*$ is in the *interior* of the feasible set, so the constraint $h(\vec{x}) \geq 0$ has no effect on the position of the $\vec{x}^*$ locally.

- $h_i(\vec{x}^*) = 0$: Such a constraint is *active*, likely indicating that if the constraint were removed $\vec{x}^*$ would no longer be optimal.
- $h_i(\vec{x}^*) > 0$: Such a constraint is *inactive*, meaning in a neighborhood of $\vec{x}^*$ if we had removed this constraint we still would have reached the same minimum.

These two cases are illustrated in Figure 10.4. While this classification will prove valuable, we do not know *a priori* which constraints will be active or inactive at $\vec{x}^*$ until we solve the optimization problem and find $\vec{x}^*$.

If all of our constraints were active, then we could change the constraint $h(\vec{x}) \geq \vec{0}$ to an equality constraint $h(\vec{x}) = \vec{0}$ without affecting the outcome of the optimization. Then, applying the equality-constrained Lagrange multiplier conditions, we could find critical points of the following Lagrange multiplier expression:

$$\Lambda(\vec{x}, \vec{\lambda}, \vec{\mu}) \equiv f(\vec{x}) - \vec{\lambda} \cdot g(\vec{x}) - \vec{\mu} \cdot h(x).$$

In reality, we no longer can say that $\vec{x}^*$ is a critical point of Λ, because inactive inequality constraints would remove terms above. Ignoring this (important!) issue for the time being, we could proceed blindly and ask for critical points of this new Λ with respect to $\vec{x}$, which satisfy the following:

$$\vec{0} = \nabla f(\vec{x}) - \sum_i \lambda_i \nabla g_i(\vec{x}) - \sum_j \mu_j \nabla h_j(\vec{x}).$$

Here, we have separated out the individual components of g and h and treated them as scalar functions to avoid complex notation.

A clever trick can extend this (currently incorrect) optimality condition to include inequality constraints. If we *define* $\mu_j \equiv 0$ whenever h_j is inactive, then the irrelevant terms are removed from the optimality conditions. In other words, we can *add* a constraint on the Lagrange multiplier above:

$$\mu_j h_j(\vec{x}) = 0.$$

With this constraint in place, we know that at least one of μ_j and $h_j(\vec{x})$ must be zero; when the constraint $h_j(\vec{x}) \geq 0$ is inactive, then μ_j must equal zero to compensate. Our first-order optimality condition still holds at critical points of the inequality-constrained problem—after adding this extra constraint.

So far, our construction has not distinguished between the constraint $h_j(\vec{x}) \geq 0$ and the constraint $h_j(\vec{x}) \leq 0$. If the constraint is inactive, it could have been dropped without affecting the outcome of the optimization locally, so we consider the case when the constraint is active. Intuitively,* in this case we expect there to be a way to decrease f by violating the constraint. Locally, the direction in which f decreases is $-\nabla f(\vec{x}^*)$ and the direction in which h_j decreases is $-\nabla h_j(\vec{x}^*)$. Thus, starting at $\vec{x}^*$ we can decrease f even more by violating the constraint $h_j(\vec{x}) \geq 0$ when $\nabla f(\vec{x}^*) \cdot \nabla h_j(\vec{x}^*) > 0$.

Products of gradients of f and h_j are difficult to manipulate. At $\vec{x}^*$, however, our first-order optimality condition tells us:

$$\nabla f(\vec{x}^*) = \sum_i \lambda_i^* \nabla g_i(\vec{x}^*) + \sum_{j \text{ active}} \mu_j^* \nabla h_j(\vec{x}^*).$$

The inactive μ_j values are zero and can be removed. We removed the $g(\vec{x}) = 0$ constraints by adding inequality constraints $g(\vec{x}) \geq \vec{0}$ and $g(\vec{x}) \leq \vec{0}$ to h; this is a mathematical convenience rather than a numerically wise maneuver.

Taking dot products with ∇h_k for any fixed k shows:

$$\sum_{j \text{ active}} \mu_j^* \nabla h_j(\vec{x}^*) \cdot \nabla h_k(\vec{x}^*) = \nabla f(\vec{x}^*) \cdot \nabla h_k(\vec{x}^*) \geq 0.$$

Vectorizing this expression shows $Dh(\vec{x}^*)Dh(\vec{x}^*)^\top \vec{\mu}^* \geq \vec{0}$. Since $Dh(\vec{x}^*)Dh(x^*)^\top$ is positive semidefinite, this implies $\vec{\mu}^* \geq \vec{0}$. Thus, the $\nabla f(\vec{x}^*) \cdot \nabla h_j(\vec{x}^*) \geq 0$ observation is equivalent to the much easier condition $\mu_j \geq 0$.

These observations can be combined and formalized to prove a first-order optimality condition for inequality-constrained minimization problems:

Theorem 10.1 (Karush-Kuhn-Tucker (KKT) conditions)**.** The vector $\vec{x}^* \in \mathbb{R}^n$ is a critical point for minimizing f subject to $g(\vec{x}) = \vec{0}$ and $h(\vec{x}) \geq \vec{0}$ when there exists $\vec{\lambda} \in \mathbb{R}^m$ and $\vec{\mu} \in \mathbb{R}^p$ such that:

- $\vec{0} = \nabla f(\vec{x}^*) - \sum_i \lambda_i \nabla g_i(\vec{x}^*) - \sum_j \mu_j \nabla h_j(\vec{x}^*)$ ("stationarity")
- $g(\vec{x}^*) = \vec{0}$ and $h(\vec{x}^*) \geq \vec{0}$ ("primal feasibility")
- $\mu_j h_j(\vec{x}^*) = 0$ for all j ("complementary slackness")
- $\mu_j \geq 0$ for all j ("dual feasibility")

When h is removed, this theorem reduces to the Lagrange multiplier criterion.

*You should not consider this discussion a formal proof, since we do not consider many boundary cases.

Example 10.6 (KKT conditions)**.** Suppose we wish to solve the following optimization (proposed by R. Israel, UBC Math 340, Fall 2006):

$$\begin{aligned} &\text{maximize } xy \\ &\text{subject to } x + y^2 \leq 2 \\ &\qquad x, y \geq 0. \end{aligned}$$

In this case we will have no λ's and three μ's. We take $f(x, y) = -xy$, $h_1(x, y) \equiv 2 - x - y^2$, $h_2(x, y) = x$, and $h_3(x, y) = y$. The KKT conditions are:

$$\begin{aligned} \text{Stationarity: } & 0 = -y + \mu_1 - \mu_2 \\ & 0 = -x + 2\mu_1 y - \mu_3 \\ \text{Primal feasibility: } & x + y^2 \leq 2 \\ & x, y \geq 0 \\ \text{Complementary slackness: } & \mu_1(2 - x - y^2) = 0 \\ & \mu_2 x = 0 \\ & \mu_3 y = 0 \\ \text{Dual feasibility: } & \mu_1, \mu_2, \mu_3 \geq 0 \end{aligned}$$

Example 10.7 (Linear programming)**.** Consider the optimization:

$$\begin{aligned} &\text{minimize}_{\vec{x}} \ \vec{b} \cdot \vec{x} \\ &\text{subject to } A\vec{x} \geq \vec{c}. \end{aligned}$$

Example 10.2 can be written this way. The KKT conditions for this problem are:

$$\begin{aligned} \text{Stationarity: } & A^\top \vec{\mu} = \vec{b} \\ \text{Primal feasibility: } & A\vec{x} \geq \vec{c} \\ \text{Complementary slackness: } & \mu_i(\vec{a}_i \cdot \vec{x} - c_i) = 0 \ \forall i, \text{ where } \vec{a}_i^\top \text{ is row } i \text{ of } A \\ \text{Dual feasibility: } & \vec{\mu} \geq \vec{0} \end{aligned}$$

As with Lagrange multipliers, we cannot assume that any $\vec{x}^*$ satisfying the KKT conditions automatically minimizes f subject to the constraints, even locally. One way to check for local optimality is to examine the Hessian of f restricted to the subspace of $\mathbb{R}^n$ in which $\vec{x}$ can move without violating the constraints. If this "reduced" Hessian is positive definite, then the optimization has reached a local minimum.

10.3 OPTIMIZATION ALGORITHMS

A careful consideration of algorithms for constrained optimization is out of the scope of our discussion. Thankfully, many stable implementations of these techniques exist, and much can be accomplished as a "client" of this software rather than rewriting it from scratch. Even so, it is useful to sketch common approaches to gain some intuition for how these libraries work.

10.3.1 Sequential Quadratic Programming (SQP)

Similar to BFGS and other methods we considered in Chapter 9, one typical strategy for constrained optimization is to approximate f, g, and h with simpler functions, solve the approximate optimization, adjust the approximation based on the latest function evaluation, and repeat.

Suppose we have a guess $\vec{x}_k$ of the solution to the constrained optimization problem. We could apply a second-order Taylor expansion to f and first-order approximation to g and h to define a next iterate as the following:

$$\begin{aligned}\vec{x}_{k+1} \equiv \vec{x}_k + \arg\min_{\vec{d}} & \left[\frac{1}{2}\vec{d}^\top H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k)\cdot\vec{d} + f(\vec{x}_k)\right] \\ \text{subject to } & g_i(\vec{x}_k) + \nabla g_i(\vec{x}_k)\cdot\vec{d} = 0 \\ & h_i(\vec{x}_k) + \nabla h_i(\vec{x}_k)\cdot\vec{d} \geq 0.\end{aligned}$$

The optimization to find $\vec{d}$ has a quadratic objective with linear constraints, which can be solved using one of many specialized algorithms; it is known as a *quadratic program*. This Taylor approximation, however, only works in a neighborhood of the optimal point. When a good initial guess $\vec{x}_0$ is unavailable, these strategies may fail.

10.3.1.1 Equality Constraints

When the only constraints are equalities and h is removed, the quadratic program for $\vec{d}$ has Lagrange multiplier optimality conditions derived as follows:

$$\begin{aligned}\Lambda(\vec{d},\vec{\lambda}) &\equiv \frac{1}{2}\vec{d}^\top H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k)\cdot\vec{d} + f(\vec{x}_k) + \vec{\lambda}^\top(g(\vec{x}_k) + Dg(\vec{x}_k)\vec{d}) \\ \implies \vec{0} = \nabla_{\vec{d}}\Lambda &= H_f(\vec{x}_k)\vec{d} + \nabla f(\vec{x}_k) + [Dg(\vec{x}_k)]^\top\vec{\lambda}.\end{aligned}$$

Combining this expression with the linearized equality constraint yields a symmetric linear system for $\vec{d}$ and $\vec{\lambda}$:

$$\begin{pmatrix} H_f(\vec{x}_k) & [Dg(\vec{x}_k)]^\top \\ Dg(\vec{x}_k) & 0 \end{pmatrix}\begin{pmatrix}\vec{d}\\ \vec{\lambda}\end{pmatrix} = \begin{pmatrix}-\nabla f(\vec{x}_k)\\ -g(\vec{x}_k)\end{pmatrix}.$$

Each iteration of sequential quadratic programming in the presence of only equality constraints can be implemented by solving this linear system to get $\vec{x}_{k+1} \equiv \vec{x}_k + \vec{d}$. This linear system is *not* positive definite, so on a large scale it can be difficult to solve. Extensions operate like BFGS for unconstrained optimization by approximating the Hessian H_f. Stability also can be improved by limiting the distance that $\vec{x}$ can move during any single iteration.

10.3.1.2 Inequality Constraints

Specialized algorithms exist for solving quadratic programs rather than general nonlinear programs that can be used for steps of SQP. One notable strategy is to keep an "active set" of constraints that are active at the minimum with respect to $\vec{d}$. The equality-constrained methods above can be applied by ignoring inactive constraints. Iterations of active-set optimization update the active set by adding violated constraints and removing those inequality constraints h_j for which $\nabla f \cdot \nabla h_j \leq 0$ as in §10.2.2.

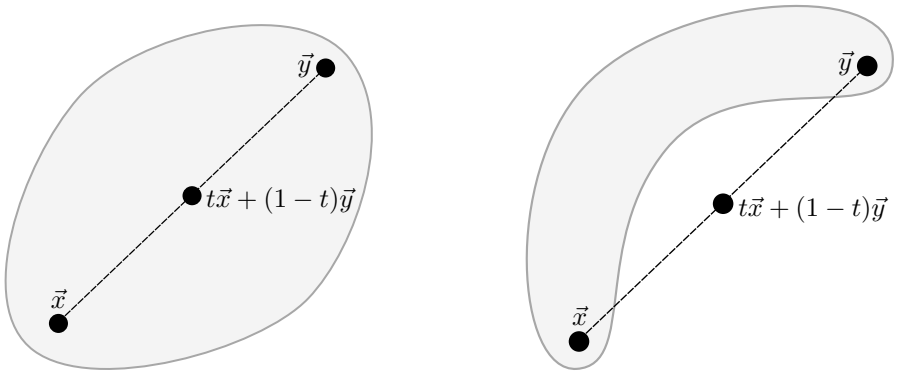


Figure 10.5 Convex and nonconvex shapes on the plane.

10.3.2 Barrier Methods

Another option for constrained minimization is to change the constraints to energy terms. For example, in the equality constrained case we could minimize an "augmented" objective as follows:

$$f_\rho(\vec{x}) = f(\vec{x}) + \rho\|g(\vec{x})\|_2^2.$$

Taking $\rho \to \infty$ will force $\|g(\vec{x})\|_2$ to be as small as possible, eventually reaching $g(\vec{x}) \approx \vec{0}$.

Barrier methods for constrained optimization apply iterative *unconstrained* optimization to f_ρ and check how well the constraints are satisfied; if they are not within a given tolerance, ρ is increased and the optimization continues using the previous iterate as a starting point. Barrier methods are simple to implement and use, but they can exhibit some pernicious failure modes. In particular, as ρ increases, the influence of f on the objective function diminishes and the Hessian of f_ρ becomes more and more poorly conditioned.

Barrier methods be constructed for inequality constraints as well as equality constraints. In this case, we must ensure that $h_i(\vec{x}) \geq 0$ for all i. Typical choices of barrier functions for inequality constraints include $1/h_i(\vec{x})$ (the "inverse barrier") and $-\log h_i(\vec{x})$ (the "logarithmic barrier").

10.4 CONVEX PROGRAMMING

The methods we have described for constrained optimization come with few guarantees on the quality of the output. Certainly they are unable to obtain global minima without a good initial guess $\vec{x}_0$, and in some cases, e.g., when Hessians near $\vec{x}^*$ are not positive definite, they may not converge at all.

There is a notable exception to this rule, which appears in many well-known optimization problems: *convex programming*. The idea here is that when f is a convex function and the feasible set itself is convex, then the optimization problem possesses a unique minimum. We considered convex functions in Definition 9.4 and now expand the class of convex problems to those containing convex constraint sets:

Definition 10.3 (Convex set)**.** A set $S \subseteq \mathbb{R}^n$ is *convex* if for any $\vec{x}, \vec{y} \in S$, the point $t\vec{x} + (1-t)\vec{y}$ is also in S for any $t \in [0, 1]$.

Intuitively, a set is convex if its boundary does not bend inward, as shown in Figure 10.5.

Example 10.8 (Circles)**.** The disc $\{\vec{x} \in \mathbb{R}^n : \|\vec{x}\|_2 \leq 1\}$ is convex, while the unit circle $\{\vec{x} \in \mathbb{R}^n : \|\vec{x}\|_2 = 1\}$ is not.

A nearly identical proof to that of Proposition 9.1 shows:

A convex function cannot have suboptimal local minima even when it is restricted to a convex domain.

If a convex objective function has two local minima, then the line of points between those minima must yield objective values less than or equal to those on the endpoints; by Definition 10.3 this entire line is feasible, completing the proof.

Strong convergence guarantees are available for convex optimization methods that guarantee finding a *global* minimum so long as f is convex and the constraints on g and h make a convex feasible set. A valuable exercise for any optimization problem is to check if it is convex, since this property can increase confidence in the output quality and the chances of success by a large factor.

A new field called *disciplined convex programming* attempts to chain together rules about convexity to generate convex optimization problems. The end user is allowed to combine convex energy terms and constraints so long as they do not violate the convexity of the final problem; the resulting objective and constraints are then provided automatically to an appropriate solver. Useful statements about convexity that can be used to construct convex programs from smaller convex building blocks include the following:

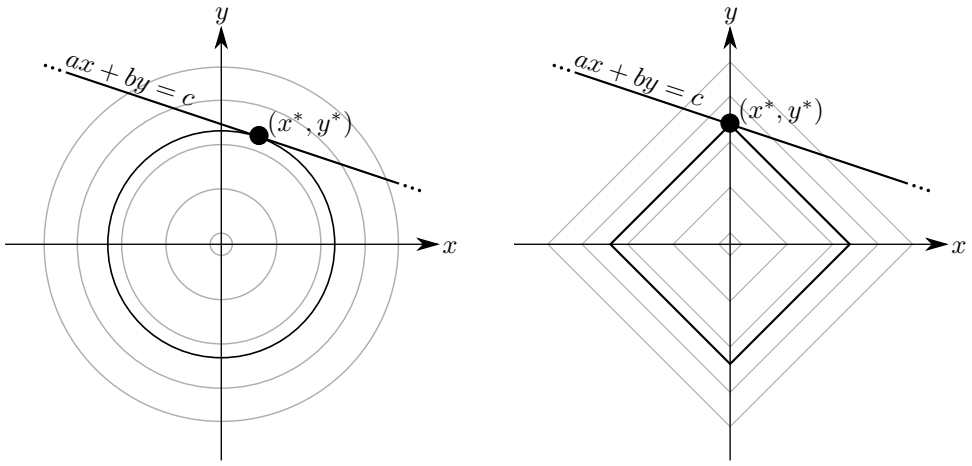
- The intersection of convex sets is convex; thus, enforcing more than one convex constraint is allowable.
- The sum of convex functions is convex.
- If f and g are convex, so is $h(\vec{x}) \equiv \max\{f(\vec{x}), g(\vec{x})\}$.
- If f is a convex function, the set $\{\vec{x} : f(\vec{x}) \leq c\}$ is convex for fixed $c \in \mathbb{R}$.

Tools such as the `CVX` library help separate implementation of convex programs from the mechanics of minimization algorithms [51, 52].

Example 10.9 (Convex programming)**.**

- The nonnegative least-squares problem in Example 10.3 is convex because $\|A\vec{x} - \vec{b}\|_2$ is a convex function of $\vec{x}$ and the set $\{\vec{x} \in \mathbb{R}^n : \vec{x} \geq \vec{0}\}$ is convex.
- Linear programs, introduced in Example 10.7, are convex because they have linear objectives and linear constraints.
- We can include $\|\vec{x}\|_1$ in a convex optimization objective, if $\vec{x}$ is an optimization variable. To do so, introduce a variable $\vec{y}$ and add constraints $y_i \geq x_i$ and $y_i \geq -x_i$ for each i. Then, $\|\vec{x}\|_1$ can be written as $\sum_i y_i$. At the minimum, we must have $y_i = |x_i|$ since we have constrained $y_i \geq |x_i|$ and might as well minimize the elements of $\vec{y}$. "Disciplined" convex libraries do such operations behind the scenes without exposing substitutions and helper variables to the end user.

Convex programming has much in common with areas of computer science theory involving *reductions* of algorithmic problems to one another. Rather than verifying NP-completeness, however, in this context we wish to use a generic solver to optimize a given objective, just like we reduced assorted problems to a linear solve in Chapter 4. There is a



(a) $p = 2$ (b) $p = 1$

Figure 10.6 On the (x, y) plane, the optimization minimizing $\|(x, y)\|_p$ subject to $ax + by = c$ has considerably different output depending on whether we choose (a) $p = 2$ or (b) $p = 1$. Level sets $\{(x, y) : \|(x, y)\|_p = c\}$ are shown in gray.

formidable pantheon of industrial-scale convex programming tools that can handle different classes of problems with varying efficiency and generality; below, we discuss some common classes. See [15, 84] for larger discussions of related topics.

10.4.1 Linear Programming

A well-studied example of convex optimization is *linear programing*, introduced in Example 10.7. Exercise 10.4 will walk through the derivation of some properties making linear programs attractive both theoretically and from an algorithmic design standpoint.

The famous *simplex algorithm*, which can be considered an active set method as in §10.3.1.2, updates the estimate of $\vec{x}^*$ using a linear solve, and checks if the active set must be updated. No Taylor approximations are needed because the objective and constraints are linear. *Interior point* linear programming algorithms such as the barrier method in §10.3.2 also are successful for these problems. Linear programs can be solved on a *huge* scale—up to millions or billions of variables!—and often appear in problems like scheduling or pricing.

One popular application of linear programming inspired by Example 10.9 provides an alternative to using pseudoinverse for underdetermined linear systems (§7.2.1). When a matrix A is underdetermined, there are many vectors $\vec{x}$ that satisfy $A\vec{x} = \vec{b}$ for a given vector $\vec{b}$. In this case, the pseudoinverse A^+ applied to $\vec{b}$ solves the following problem:

$$\text{Pseudoinverse} \left\{ \begin{array}{ll} \text{minimize}_{\vec{x}} & \|\vec{x}\|_2 \\ \text{subject to} & A\vec{x} = \vec{b}. \end{array} \right.$$

Using linear programs, we can solve a slightly different system:

$$L_1 \text{ minimization} \left\{ \begin{array}{ll} \text{minimize}_{\vec{x}} & \|\vec{x}\|_1 \\ \text{subject to} & A\vec{x} = \vec{b}. \end{array} \right.$$

All we have done here is replace the norm $\|\cdot\|_2$ with a different norm $\|\cdot\|_1$.

Why does this one-character change make a significant difference in the output $\vec{x}$? Consider the two-dimensional instance of this problem shown in Figure 10.6, which minimizes

$\|(x, y)\|_p$ for $p = 2$ (pseudoinverse) and $p = 1$ (linear program). In the $p = 2$ case (a), we are minimizing $x^2 + y^2$, which has circular level sets; the optimal (x^*, y^*) subject to the constraints is in the interior of the first quadrant. In the $p = 1$ case (b), we are minimizing $|x| + |y|$, which has diamond-shaped level sets; this makes $x^* = 0$ since the outer points of the diamond align with the x and y axes, a more *sparse* solution.

More generally, the use of the norm $\|\vec{x}\|_2$ indicates that no single element x_i of $\vec{x}$ should have a large value; this regularization tends to favor vectors $\vec{x}$ with lots of small nonzero values. On the other hand, $\|\vec{x}\|_1$ does not care if a single element of $\vec{x}$ has a large value so long as the sum of all the elements' absolute values is small. As we have illustrated in the two-dimensional case, this type of regularization can produce *sparse* vectors $\vec{x}$, with elements that are exactly zero.

This type of regularization using $\|\cdot\|_1$ is fundamental in the field of *compressed sensing*, which solves underdetermined signal processing problems with the additional assumption that the output should be sparse. This assumption makes sense in many contexts where sparse solutions of $A\vec{x} = \vec{b}$ imply that many columns of A are irrelevant [37].

A minor extension of linear programming is to keep using linear inequality constraints but introduce convex quadratic terms to the objective, changing the optimization in Example 10.7 to:

$$\begin{aligned}&\text{minimize}_{\vec{x}}\ \vec{b}\cdot\vec{x} + \vec{x}^\top M\vec{x}\\ &\text{subject to } A\vec{x} \geq \vec{c}.\end{aligned}$$

Here, M is an $n \times n$ positive semidefinite matrix. With this machinery, we can provide an alternative to Tikhonov regularization from §4.1.3:

$$\min_{\vec{x}} \|A\vec{x} - \vec{b}\|_2^2 + \alpha\|\vec{x}\|_1.$$

This "lasso" regularizer also promotes sparsity in $\vec{x}$ while solving $A\vec{x} \approx \vec{b}$, but it does not enforce $A\vec{x} = \vec{b}$ exactly. It is useful when A or $\vec{b}$ is noisy and we prefer sparsity of $\vec{x}$ over solving the system exactly [119].

10.4.2 Second-Order Cone Programming

A *second-order cone program* (SOCP) is a convex optimization problem taking the following form [15]:

$$\begin{aligned}&\text{minimize}_{\vec{x}}\ \vec{b}\cdot\vec{x}\\ &\text{subject to } \|A_i\vec{x} - \vec{b}_i\|_2 \leq d_i + \vec{c}_i\cdot\vec{x} \text{ for all } i = 1, \ldots, k.\end{aligned}$$

Here, we use matrices $A_1, \ldots, A_k$, vectors $\vec{b}_1, \ldots, \vec{b}_k$, vectors $\vec{c}_1, \ldots, \vec{c}_k$, and scalars $d_1, \ldots, d_k$ to specify the k constraints. These "cone constraints" will allow us to pose a broader set of convex optimization problems.

One non-obvious application of second-order cone programming explained in [83] appears when we wish to solve the least-squares problem $A\vec{x} \approx \vec{b}$, but we do not know the elements of A exactly. For instance, A might have been constructed from data we have measured experimentally (see §4.1.2 for an example in least-squares regression).

Take $\vec{a}_i^\top$ to be the i-th row of A. Then, the least-squares problem $A\vec{x} \approx \vec{b}$ can be understood as minimizing $\sum_i(\vec{a}_i\cdot\vec{x} - b_i)^2$ over $\vec{x}$. If we do not know A exactly, however, we might allow each $\vec{a}_i$ to vary somewhat before solving least-squares. In particular, maybe we

think that $\vec{a}_i$ is an approximation of some unknown $\vec{a}_i^0$ satisfying $\|\vec{a}_i^0 - \vec{a}_i\|_2 \leq \varepsilon$ for some fixed $\varepsilon > 0$.

To make least-squares robust to this model of error, we can choose $\vec{x}$ to thwart an adversary picking the worst possible $\vec{a}_i^0$. Formally, we solve the following "minimax" problem:

$$\text{minimize}_{\vec{x}} \begin{bmatrix} \max_{\{\vec{a}_i^0\}} & \sum_i (\vec{a}_i^0 \cdot \vec{x} - b_i)^2 \\ \text{subject to} & \|\vec{a}_i^0 - \vec{a}_i\|_2 \leq \varepsilon \text{ for all } i \end{bmatrix}.$$

That is, we want to choose $\vec{x}$ so that the least-squares energy with the *worst* possible unknowns $\vec{a}_i^0$ satisfying $\|\vec{a}_i^0 - \vec{a}_i\|_2 \leq \varepsilon$ still is small. It is far from evident that this complicated optimization problem is solvable using SOCP machinery, but after some simplification we will manage to write it in the standard SOCP form above.

If we define $\delta\vec{a}_i \equiv \vec{a}_i - \vec{a}_i^0$, then our optimization becomes:

$$\text{minimize}_{\vec{x}} \begin{bmatrix} \max_{\{\delta\vec{a}_i\}} & \sum_i (\vec{a}_i \cdot \vec{x} + \delta\vec{a}_i \cdot \vec{x} - b_i)^2 \\ \text{subject to} & \|\delta\vec{a}_i\|_2 \leq \varepsilon \text{ for all } i \end{bmatrix}.$$

When maximizing over $\delta\vec{a}_i$, each term of the sum over i is independent. Hence, we can solve the inner maximization for one $\delta\vec{a}_i$ at a time. Peculiarly, if we maximize an absolute value rather than a sum (usually we go in the other direction!), we can find a closed-form solution to the optimization for $\delta\vec{a}_i$ for a single fixed i:

$$\begin{aligned}
\max_{\|\delta\vec{a}_i\|_2 \leq \varepsilon} |\vec{a}_i \cdot \vec{x} + \delta\vec{a}_i \cdot \vec{x} - b_i| &= \max_{\|\delta\vec{a}_i\|_2 \leq \varepsilon} \max\{\vec{a}_i \cdot \vec{x} + \delta\vec{a}_i \cdot \vec{x} - b_i, -\vec{a}_i \cdot \vec{x} - \delta\vec{a}_i \cdot \vec{x} + b_i\} \\
&\qquad \text{since } |x| = \max\{x, -x\} \\
&= \max\left\{ \max_{\|\delta\vec{a}_i\|_2 \leq \varepsilon} [\vec{a}_i \cdot \vec{x} + \delta\vec{a}_i \cdot \vec{x} - b_i], \right. \\
&\qquad \left. \max_{\|\delta\vec{a}_i\|_2 \leq \varepsilon} [-\vec{a}_i \cdot \vec{x} - \delta\vec{a}_i \cdot \vec{x} + b_i] \right\} \\
&\qquad \text{after changing the order of the maxima} \\
&= \max\{\vec{a}_i \cdot \vec{x} + \varepsilon\|\vec{x}\|_2 - b_i, -\vec{a}_i \cdot \vec{x} + \varepsilon\|\vec{x}\|_2 + b_i\} \\
&= |\vec{a}_i \cdot \vec{x} - b_i| + \varepsilon\|\vec{x}\|_2.
\end{aligned}$$

After this simplification, our optimization for $\vec{x}$ becomes:

$$\text{minimize}_{\vec{x}} \sum_i (|\vec{a}_i \cdot \vec{x} - b_i| + \varepsilon\|\vec{x}\|_2)^2.$$

This minimization can be written as a second-order cone problem:

$$\begin{aligned}
\text{minimize}_{s,\vec{t},\vec{x}} \quad & s \\
\text{subject to} \quad & \|\vec{t}\|_2 \leq s \\
& (\vec{a}_i \cdot \vec{x} - b_i) + \varepsilon\|\vec{x}\|_2 \leq t_i \ \forall i \\
& -(\vec{a}_i \cdot \vec{x} - b_i) + \varepsilon\|\vec{x}\|_2 \leq t_i \ \forall i.
\end{aligned}$$

In this optimization, we have introduced two extra variables s and $\vec{t}$. Since we wish to minimize s with the constraint $\|\vec{t}\|_2 \leq s$, we are effectively minimizing the norm of $\vec{t}$. The last two constraints ensure that each element of $\vec{t}$ satisfies $t_i = |\vec{a}_i \cdot \vec{x} - b_i| + \varepsilon\|\vec{x}\|_2$.

This type of regularization provides yet another variant of least-squares. In this case, rather than being robust to near-singularity of A, we have incorporated an error model directly into our formulation allowing for mistakes in measuring rows of A. The parameter ε controls sensitivity to the elements of A in a similar fashion to the weight α of Tikhonov or L_1 regularization.

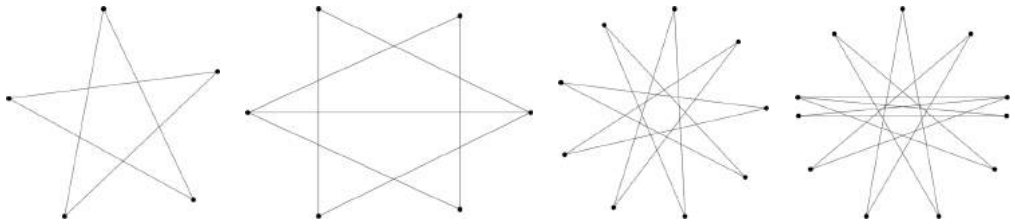


Figure 10.7 Examples of graphs laid out via semidefinite embedding.

10.4.3 Semidefinite Programming

Suppose A and B are $n \times n$ positive semidefinite matrices; we will notate this as $A, B \succeq 0$. Take $t \in [0, 1]$. Then, for any $\vec{x} \in \mathbb{R}^n$ we have:

$$\vec{x}^\top (tA + (1-t)B)\vec{x} = t\vec{x}^\top A\vec{x} + (1-t)\vec{x}^\top B\vec{x} \geq 0,$$

where the inequality holds by semidefiniteness of A and B. This proof verifies a surprisingly useful fact:

The set of positive semidefinite matrices is convex.

Hence, if we are solving optimization problems for a matrix A, we safely can add constraints $A \succeq 0$ without affecting convexity.

Algorithms for *semidefinite programming* optimize convex objectives with the ability to add constraints that matrix-valued variables must be positive (or negative) semidefinite. More generally, semidefinite programming machinery can include *linear matrix inequality* (LMI) constraints of the form:

$$x_1 A_1 + x_2 A_2 + \cdots + x_k A_k \succeq 0,$$

where $\vec{x} \in \mathbb{R}^k$ is an optimization variable and the matrices A_i are fixed.

As an example of semidefinite programming, we will sketch a technique known as *semidefinite embedding* from graph layout and manifold learning [130]. Suppose we are given a graph (V, E) consisting of a set of vertices $V = \{v_1, \ldots, v_k\}$ and a set of edges $E \subseteq V \times V$. For some fixed n, the semidefinite embedding method computes positions $\vec{x}_1, \ldots, \vec{x}_k \in \mathbb{R}^n$ for the vertices, so that vertices connected by edges are nearby in the embedding with respect to Euclidean distance $\|\cdot\|_2$; some examples are shown in Figure 10.7.

If we already have computed $\vec{x}_1, \ldots, \vec{x}_k$, we can construct a Gram matrix $G \in \mathbb{R}^{k \times k}$ satisfying $G_{ij} = \vec{x}_i \cdot \vec{x}_j$. G is a matrix of inner products and hence is symmetric and positive semidefinite. We can measure the squared distance from $\vec{x}_i$ to $\vec{x}_j$ using G:

$$\begin{aligned}\|\vec{x}_i - \vec{x}_j\|_2^2 &= (\vec{x}_i - \vec{x}_j) \cdot (\vec{x}_i - \vec{x}_j) \\ &= \|\vec{x}_i\|_2^2 - 2\vec{x}_i \cdot \vec{x}_j + \|\vec{x}_j\|_2^2 \\ &= G_{ii} - 2G_{ij} + G_{jj}.\end{aligned}$$

Similarly, suppose we wish the center of mass $\frac{1}{k}\sum_i \vec{x}_i$ to be $\vec{0}$, since shifting the embedding of the graph does not have a significant effect on its layout. We alternatively can write $\|\sum_i \vec{x}_i\|_2^2 = 0$ and can express this condition in terms of G:

$$0 = \left\| \sum_i \vec{x}_i \right\|_2^2 = \left(\sum_i \vec{x}_i \right) \cdot \left(\sum_i \vec{x}_i \right) = \sum_{ij} \vec{x}_i \cdot \vec{x}_j = \sum_{ij} G_{ij}.$$

Finally, we might wish that our embedding of the graph is relatively compact or small. One way to do this would be to minimize $\sum_i \|\vec{x}_i\|_2^2 = \sum_i G_{ii} = \text{Tr}(G)$.

The semidefinite embedding technique turns these observations on their head, optimizing for the Gram matrix G directly rather than the positions $\vec{x}_i$ of the vertices. Making use of the observations above, semidefinite embedding solves the following optimization problem:

$$\begin{array}{rl} \text{minimize}_{G \in \mathbb{R}^{k \times k}} & \text{Tr}(G) \\ \text{subject to} & G = G^\top \\ & G \succeq 0 \\ & G_{ii} - 2G_{ij} + G_{jj} = 1 \ \forall (v_i, v_j) \in E \\ & \sum_{ij} G_{ij} = 0. \end{array}$$

This optimization for G is motivated as follows:

- The objective asks that the embedding of the graph is compact by minimizing the sum of squared norms $\sum_i \|\vec{x}_i\|_2^2$.
- The first two constraints require that the Gram matrix is symmetric and positive definite.
- The third constraint requires that the embeddings of any two adjacent vertices in the graph have distance one.
- The final constraint centers the full embedding about the origin.

We can use semidefinite programming to solve this optimization problem for G. Then, since G is symmetric and positive semidefinite, we can use the Cholesky factorization (§4.2.1) or the eigenvector decomposition (§6.2) of G to write $G = X^\top X$ for some matrix $X \in \mathbb{R}^{k \times k}$. Based on the discussion above, the columns of X are an embedding of the vertices of the graph into $\mathbb{R}^k$ where all the edges in the graph have length one, the center of mass is the origin, and the total square norm of the positions is minimized.

We set out to embed the graph into $\mathbb{R}^n$ rather than $\mathbb{R}^k$, and generally $n \leq k$. To compute a lower-dimensional embedding that *approximately* satisfies the constraints, we can decompose $G = X^\top X$ using its eigenvectors; then, we remove $k - n$ eigenvectors with eigenvalues closest to zero. This operation is exactly the low-rank approximation of G via SVD given in §7.2.2. This final step provides an embedding of the graph into $\mathbb{R}^n$.

A legitimate question about the semidefinite embedding is how the optimization for G interacts with the low-rank eigenvector approximation applied in post-processing. In many well-known cases, the solution of semidefinite optimizations like the one above yield *low-rank* or nearly low-rank matrices whose lower-dimensional approximations are close to the original; a formalized version of this observation justifies the approximation. We already explored such a justification in Exercise 7.7, since the nuclear norm of a symmetric positive semidefinite matrix is its trace.

10.4.4 Integer Programs and Relaxations

Our final application of convex optimization is—surprisingly—to a class of *highly* non-convex problems: Ones with integer variables. In particular, an *integer program* is an optimization in which one or more variables is constrained to be an integer rather than a real number. Within this class, two well-known subproblems are *mixed-integer programming*, in which some variables are continuous while others are integers, and *zero-one programming*, where the variables take Boolean values in $\{0, 1\}$.

Example 10.10 (3-SAT)**.** We can define the following operations from Boolean algebra for binary variables $U, V \in \{0, 1\}$:

U	V	$\neg U$ ("not U")	$\neg V$ ("not V")	$U \wedge V$ ("U and V")	$U \vee V$ ("U or V")
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

We can convert Boolean satisfiability problems into integer programs using a few steps. For example, we can express the "not" operation algebraically using $\neg U = 1 - U$. Similarly, suppose we wish to find U, V satisfying $(U \vee \neg V) \wedge (\neg U \vee V)$. Then, U and V as integers satisfy the following constraints:

$$\begin{aligned} U + (1 - V) &\geq 1 && (U \vee \neg V) \\ (1 - U) + V &\geq 1 && (\neg U \vee V) \\ U, V &\in \mathbb{Z} && \text{(integer constraint)} \\ 0 \leq U, V &\leq 1 && \text{(Boolean variables)} \end{aligned}$$

As demonstrated in Example 10.10, integer programs encode a wide class of discrete problems, including many that are known to be NP-hard. For this reason, we cannot expect to solve them exactly with convex optimization; doing so would settle a long-standing question of theoretical computer science by showing "$P = NP$." We can, however, use convex optimization to find approximate solutions to integer programs.

If we write a discrete problem like Example 10.10 as an optimization, we can *relax* the constraint keeping variables in $\mathbb{Z}$ and allow them to be in $\mathbb{R}$ instead. Such a relaxation can yield invalid solutions, e.g., Boolean variables that take on values like 0.75. So, after solving the relaxed problem, one of many strategies can be used to generate an integer approximation of the solution. For example, non-integral variables can be *rounded* to the closest integer, at the risk of generating outputs that are suboptimal or violate the constraints. Alternatively, a slower but potentially more effective method iteratively rounds one variable at a time, adds a constraint fixing the value of that variable, and re-optimizes the objective subject to the new constraint.

Many difficult discrete problems can be reduced to integer programs, from satisfiability problems like the one in Example 10.10 to the traveling salesman problem. These reductions should indicate that the design of effective integer programming algorithms is challenging even in the approximate case. State-of-the-art convex relaxation methods for integer programming, however, are fairly effective for a large class of problems, providing a remarkably general piece of machinery for approximating solutions to problems for which it may be difficult or impossible to design a discrete algorithm. Many open research problems involve designing effective integer programming methods and understanding potential relaxations; this work provides a valuable and attractive link between continuous and discrete mathematics.

10.5 EXERCISES

10.1 Prove the following statement from §10.4: If f is a convex function, the set $\{\vec{x} : f(\vec{x}) \leq c\}$ is convex.

10.2 The standard deviation of k values $x_1, \ldots, x_k$ is

$$\sigma(x_1, \ldots, x_k) \equiv \sqrt{\frac{1}{k}\sum_{i=1}^{k}(x_i - \mu)^2},$$

where $\mu \equiv \frac{1}{k}\sum_i x_i$. Show that σ is a convex function of $x_1, \ldots, x_k$.

10.3 Some properties of second-order cone programming:

(a) Show that the *Lorentz cone* $\{\vec{x} \in \mathbb{R}^n, c \in \mathbb{R} : \|\vec{x}\|_2 \leq c\}$ is convex.

(b) Use this fact to show that the second-order cone program in §10.4.2 is convex.

(c) Show that second-order cone programming can be used to solve linear programs.

10.4 In this problem we will study *linear programming* in more detail.

(a) A linear program in "standard form" is given by:

$$\begin{array}{ll} \text{minimize}_{\vec{x}} & \vec{c}^\top \vec{x} \\ \text{subject to} & A\vec{x} = \vec{b} \\ & \vec{x} \geq \vec{0}. \end{array}$$

Here, the optimization is over $\vec{x} \in \mathbb{R}^n$; the remaining variables are constants $A \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$, and $\vec{c} \in \mathbb{R}^n$. Find the KKT conditions of this system.

(b) Suppose we add a constraint of the form $\vec{v}^\top \vec{x} \leq d$ for some fixed $\vec{v} \in \mathbb{R}^n$ and $d \in \mathbb{R}$. Explain how such a constraint can be added while keeping a linear program in standard form.

(c) The "dual" of this linear program is another optimization:

$$\begin{array}{ll} \text{maximize}_{\vec{y}} & \vec{b}^\top \vec{y} \\ \text{subject to} & A^\top \vec{y} \leq \vec{c}. \end{array}$$

Assuming that the primal and dual have exactly one stationary point, show that the optimal value of the primal and dual objectives coincide.
Hint: Show that the KKT multipliers of one problem can be used to solve the other.
Note: This property is called "strict duality." The famous simplex algorithm for solving linear programs maintains estimates of $\vec{x}$ and $\vec{y}$, terminating when $\vec{c}^\top \vec{x}^* - \vec{b}^\top \vec{y}^* = 0$.

10.5 Suppose we take a grayscale photograph of size $n \times m$ and represent it as a vector $\vec{v} \in \mathbb{R}^{nm}$ of values in $[0, 1]$. We used the wrong lens, however, and our photo is blurry! We wish to use *deconvolution* machinery to undo this effect.

(a) Find the KKT conditions for the following optimization problem:

$$\begin{array}{ll} \text{minimize}_{\vec{x} \in \mathbb{R}^{nm}} & \|A\vec{x} - \vec{b}\|_2^2 \\ \text{subject to} & 0 \leq x_i \leq 1 \ \forall i \in \{1, \ldots, nm\}. \end{array}$$

(b) Suppose we are given a matrix $G \in \mathbb{R}^{nm \times nm}$ taking sharp images to blurry ones. Propose an optimization in the form of (a) for recovering a sharp image from our blurry $\vec{v}$.

(c) We do not know the operator G, making the model in (b) difficult to use. Suppose, however, that for each $r \geq 0$ we can write a matrix $G_r \in \mathbb{R}^{nm \times nm}$ approximating a blur with radius r. Using the same camera, we now take k *pairs* of photos $(\vec{v}_1, \vec{w}_1), \ldots, (\vec{v}_k, \vec{w}_k)$, where $\vec{v}_i$ and $\vec{w}_i$ are of the same scene but $\vec{v}_i$ is blurry (taken using the same lens as our original bad photo) and $\vec{w}_i$ is sharp. Propose a nonlinear optimization for approximating r using this data.

[DH] 10.6 ("Fenchel duality," adapted from [10]) Let $f(\vec{x})$ be a convex function on $\mathbb{R}^n$ that is *proper*. This means that f accepts vectors from $\mathbb{R}^n$ or whose coordinates may (individually) be $\pm\infty$ and returns a real scalar in $\mathbb{R} \cup \{\infty\}$ with at least one $f(\vec{x}_0)$ taking a non-infinite value. Under these assumptions, the *Fenchel dual* of f at $\vec{y} \in \mathbb{R}^n$ is defined to be the function

$$f^*(\vec{y}) \equiv \sup_{\vec{x} \in \mathbb{R}^n} (\vec{x} \cdot \vec{y} - f(\vec{x})).$$

Fenchel duals are used to study properties of convex optimization problems in theory and practice.

(a) Show that f^* is convex.

(b) Derive the *Fenchel-Young inequality*:

$$f(\vec{x}) + f^*(\vec{y}) \geq \vec{x} \cdot \vec{y}.$$

(c) The *indicator function* of a subset $A \in \mathbb{R}^n$ is given by

$$\chi_A(\vec{x}) \equiv \begin{cases} 0 & \text{if } \vec{x} \in A \\ \infty & \text{otherwise.} \end{cases}$$

With this definition in mind, determine the Fenchel dual of $f(\vec{x}) = \vec{c} \cdot \vec{x}$, where $\vec{c} \in \mathbb{R}^n$.

(d) What is the Fenchel dual of the linear function $f(x) = ax + b$?

(e) Show that $f(\vec{x}) = \frac{1}{2}\|\vec{x}\|_2^2$ is *self-dual*, meaning $f = f^*$.

(f) Suppose $p, q \in (1, \infty)$ satisfy $\frac{1}{p} + \frac{1}{q} = 1$. Show that the Fenchel dual of $f(x) = \frac{1}{p}|x|^p$ is $f^*(y) = \frac{1}{q}|y|^q$. Use this result along with previous parts of this problem to derive Hölder's inequality

$$\sum_k |u_k v_k| \leq \left(\sum_k |u_k|^p \right)^{1/p} \left(\sum_k |v_k|^q \right)^{1/q},$$

for all $\vec{u}, \vec{v} \in \mathbb{R}^n$.

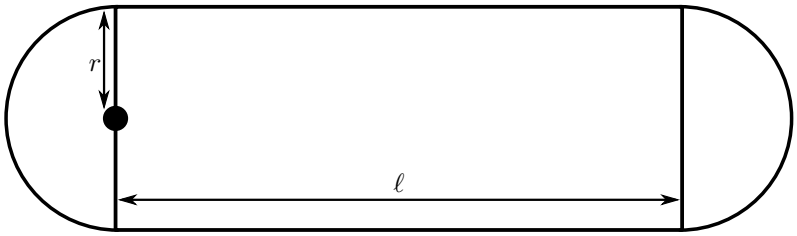


Figure 10.8 Notation for Exercise 10.7.

[SC] 10.7 A monomial is a function of the form $f(\vec{x}) = cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}$, where each $a_i \in \mathbb{R}$ and $c > 0$. We define a *posynomial* as a sum of one or more monomials:

$$f(\vec{x}) = \sum_{k=1}^{K} c_k x_1^{a_{k_1}} x_2^{a_{k_2}} \cdots x_n^{a_{k_n}}.$$

Geometric programs are optimization problems taking the following form:

$$\begin{array}{ll} \text{minimize}_{\vec{x}} & f_0(\vec{x}) \\ \text{subject to} & f_i(\vec{x}) \leq 1 \,\forall i \in \{1,\ldots,m\} \\ & g_i(\vec{x}) = 1 \,\forall i \in \{1,\ldots,p\}, \end{array}$$

where the functions f_i are posynomials and the functions g_i are monomials.

(a) Suppose you are designing a slow-dissolving medicinal capsule. The capsule looks like a cylinder with hemispherical ends, illustrated in Figure 10.8. To ensure that the capsule dissolves slowly, you need to minimize its surface area.
The cylindrical portion of the capsule must have volume larger than or equal to V to ensure that it can hold the proper amount of medicine. Also, because the capsule is manufactured as two halves that slide together, to ensure that the capsule will not break, the length ℓ of its cylindrical portion must be at least $\ell_{\min}$. Finally, due to packaging limitations, the total length of the capsule must be no larger than C.
Write the corresponding minimization problem and argue that it is a geometric program.

(b) Transform the problem from Exercise 10.7a into a convex programming problem. *Hint:* Consider the substitution $y_i = \log x_i$.

10.8 The *cardinality function* $\|\cdot\|_0$ computes the number of nonzero elements of $\vec{x} \in \mathbb{R}^n$:

$$\|\vec{x}\|_0 = \sum_{i=1}^{n} \begin{cases} 1 & x_i \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

(a) Show that $\|\cdot\|_0$ is *not* a norm on $\mathbb{R}^n$, but that it is connected to L_p norms by the relationship

$$\|\vec{x}\|_0 = \lim_{p \to 0^+} \sum_{i=1}^{n} |x_i|^p.$$

(b) Suppose we wish to solve an underdetermined system of equations $A\vec{x} = \vec{b}$. One alternative to SVD-based approaches or Tikhonov regularizations is *cardinality minimization*:

$$\begin{array}{rl} \min_{\vec{x}\in\mathbb{R}^n} & \|\vec{x}\|_0 \\ \text{subject to} & A\vec{x} = \vec{b} \\ & \|\vec{x}\|_\infty \leq R. \end{array}$$

Rewrite this optimization in the form

$$\begin{array}{rl} \min_{\vec{x},\vec{z}} & \|\vec{z}\|_1 \\ \text{subject to} & \vec{z} \in \{0,1\}^n \\ & \vec{x}, \vec{z} \in \mathcal{C}, \end{array}$$

where $\mathcal{C}$ is some convex set [15].

(c) Show that relaxing the constraint $\vec{z} \in \{0,1\}^n$ to $\vec{z} \in [0,1]^n$ lower-bounds the original problem. Propose a heuristic for the $\{0,1\}$ problem based on this relaxation.

10.9 ("Grasping force optimization;" adapted from [83]) Suppose we are writing code to control a robot hand with n fingers grasping a rigid object. Each finger i is controlled by a motor that outputs nonnegative torque t_i.

The force $\vec{F}_i$ imparted by each finger onto the object can be decomposed into two orthogonal parts as $\vec{F}_i = \vec{F}_{ni} + \vec{F}_{si}$, a normal force $\vec{F}_{ni}$ and a tangential friction force $\vec{F}_{si}$:

$$\textbf{Normal force: } \vec{F}_{ni} = c_i t_i \vec{v}_i = (\vec{v}_i^\top \vec{F}_i)\vec{v}_i$$
$$\textbf{Friction force: } \vec{F}_{si} = (I_{3\times 3} - \vec{v}_i\vec{v}_i^\top)\vec{F}_i, \text{ where } \|\vec{F}_{si}\|_2 \leq \mu\|F_{ni}\|_2$$

Here, $\vec{v}_i$ is a (fixed) unit vector normal to the surface at the point of contact of finger i. The value c_i is a constant associated with finger i. Additionally, the object experiences a gravitational force in the downward direction given by $\vec{F}_g = m\vec{g}$.

For the object to be grasped firmly in place, the sum of the forces exerted by all fingers must be $\vec{0}$. Show how to minimize the total torque outputted by the motors while firmly grasping the object using a second-order cone program.

10.10 Show that when $\vec{c}_i = \vec{0}$ for all i in the second-order cone program of §10.4.2, the optimization problem can be solved as a convex quadratic program with quadratic constraints.

10.11 (Suggested by Q. Huang) Suppose we know

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & x \\ 1 & x & 1 \end{pmatrix} \succeq 0.$$

What can we say about x?

[SC] 10.12 We can modify the gradient descent algorithm for minimizing $f(\vec{x})$ to account for linear equality constraints $A\vec{x} = \vec{b}$.

(a) Assuming we choose $\vec{x}_0$ satisfying the equality constraint, propose a modification to gradient descent so that each iterate $\vec{x}_k$ satisfies $A\vec{x}_k = \vec{b}$.
Hint: The gradient $\nabla f(\vec{x})$ may point in a direction that could violate the constraint.

(b) Briefly justify why the modified gradient descent algorithm should reach a local minimum of the constrained optimization problem.

(c) Suppose rather than $A\vec{x} = \vec{b}$ we have a nonlinear constraint $g(\vec{x}) = \vec{0}$. Propose a modification of your strategy from Exercise 10.12a maintaining this new constraint approximately. How is the modification affected by the choice of step sizes?

10.13 Show that linear programming and second-order cone programming are special cases of semidefinite programming.