11 One-dimensional Optimization

Lab Objective: *Most mathematical optimization problems involve estimating the minimizer(s) of a scalar-valued function. Many algorithms for optimizing functions with a high-dimensional domain depend on routines for optimizing functions of a single variable. There are many techniques for optimization in one dimension, each with varying degrees of precision and speed. In this lab, we implement the golden section search method, Newton's method, and the secant method, then apply them to the backtracking problem.*

Golden Section Search

A function $f : [a, b] \to \mathbb{R}$ satisfies the *unimodal property* if it has exactly one local minimum and is monotonic on either side of the minimizer. In other words, f decreases from a to its minimizer x^*, then increases up to b (see Figure 11.1). The *golden section search* method optimizes a unimodal function f by iteratively defining smaller and smaller intervals containing the unique minimizer x^*. This approach is especially useful if the function's derivative does not exist, is unknown, or is very costly to compute.

By definition, the minimizer x^* of f must lie in the interval $[a, b]$. To shrink the interval around x^*, we test the following strategically chosen points.

$$\tilde{a} = b - \frac{b-a}{\varphi} \qquad \tilde{b} = a + \frac{b-a}{\varphi}$$

Here $\varphi = \frac{1+\sqrt{5}}{2}$ is the *golden ratio*. At each step of the search, $[a, b]$ is refined to either $[a, \tilde{b}]$ or $[\tilde{a}, b]$, called the *golden sections*, depending on the following criteria.

If $f(\tilde{a}) < f(\tilde{b})$, then since f is unimodal, it must be increasing in a neighborhood of $\tilde{b}$. The unimodal property also guarantees that f must be increasing on $[\tilde{b}, b]$ as well, so $x^* \in [a, \tilde{b}]$ and we set $b = \tilde{b}$. By similar reasoning, if $f(\tilde{a}) > f(\tilde{b})$, then $x^* \in [\tilde{a}, b]$ and we set $a = \tilde{a}$. If, however, $f(\tilde{a}) = f(\tilde{b})$, then the unimodality of f does not guarantee anything about where the minimizer lies. Assuming either $x^* \in [a, \tilde{b}]$ or $x^* \in [\tilde{a}, b]$ allows the iteration to continue, but the method is no longer guaranteed to converge to the local minimum.

At each iteration, the length of the search interval is divided by φ. The method therefore converges linearly, which is somewhat slow. However, the idea is simple and each step is computationally inexpensive.

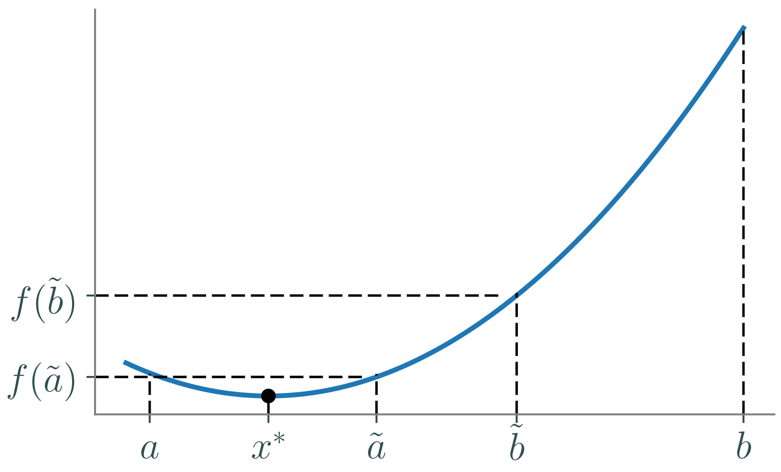

Figure 11.1: The unimodal $f : [a, b] \to \mathbb{R}$ can be minimized with a golden section search. For the first iteration, $f(\tilde{a}) < f(\tilde{b})$, so $x^* \in [a, \tilde{b}]$. New values of $\tilde{a}$ and $\tilde{b}$ are then calculated from this new, smaller interval.

Algorithm 11.1 The Golden Section Search

```
procedure GOLDEN_SECTION($f$, $a$, $b$, tol, maxiter)
    $x_0 \leftarrow (a + b)/2$                ▷ Set the initial minimizer approximation as the interval midpoint.
    $\varphi = (1 + \sqrt{5})/2$
    for $i = 1, 2, \dots,$ maxiter do                ▷ Iterate only maxiter times at most.
        $c \leftarrow (b - a)/\varphi$
        $\tilde{a} \leftarrow b - c$
        $\tilde{b} \leftarrow a + c$
        if $f(\tilde{a}) \leq f(\tilde{b})$ then                ▷ Get new boundaries for the search interval.
            $b \leftarrow \tilde{b}$
        else
            $a \leftarrow \tilde{a}$
        $x_1 \leftarrow (a + b)/2$                ▷ Set the minimizer approximation as the interval midpoint.
        if $|x_0 - x_1| <$ tol then
            break                ▷ Stop iterating if the approximation stops changing enough.
        $x_0 \leftarrow x_1$
    return $x_1$
```

Problem 1. Write a function that accepts a function $f : \mathbb{R} \to \mathbb{R}$, interval limits a and b, a stopping tolerance `tol`, and a maximum number of iterations `maxiter`. Use Algorithm 11.1 to implement the golden section search. Return the approximate minimizer x^*, whether or not the algorithm converged (`true` or `false`), and the number of iterations computed.

Test your function by minimizing $f(x) = e^x - 4x$ on the interval $[0, 3]$, then plotting the function and the computed minimizer together. Also compare your results to SciPy's golden section search, `scipy.optimize.golden()`.

```
>>> from scipy import optimize as opt
>>> import numpy as np

>>> f = lambda x : np.exp(x) - 4*x
>>> opt.golden(f, brack=(0,3), tol=.001)
1.3862578679031485              # ln(4) is the minimizer.
```

Newton's Method

Newton's method is an important root-finding algorithm that can also be used for optimization. Given $f : \mathbb{R} \to \mathbb{R}$ and a good initial guess x_0, the sequence $(x_k)_{k=1}^{\infty}$ generated by the recursive rule

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

converges to a point $\bar{x}$ satisfying $f(\bar{x}) = 0$. The first-order necessary conditions from elementary calculus state that if f is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of f' is a way to identify potential minima or maxima of f. Specifically, starting with an initial guess x_0, set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{11.1}$$

and iterate until $|x_k - x_{k-1}|$ is satisfactorily small. Note that this procedure does not use the actual function f at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

Each step of (11.1) can be thought of approximating the objective function f by a quadratic function q and finding its unique extrema. That is, we first approximate f with its second-degree Taylor polynomial centered at x_k.

$$q(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$$

This quadratic function satisfies $q(x_k) = f(x_k)$ and matches f fairly well close to x_k. Thus the optimizer of q is a reasonable guess for an optimizer of f. To compute that optimizer, solve $q'(x) = 0$.

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \qquad \Longrightarrow \qquad x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (11.1) using x_{k+1} for x. See Figure 11.2.

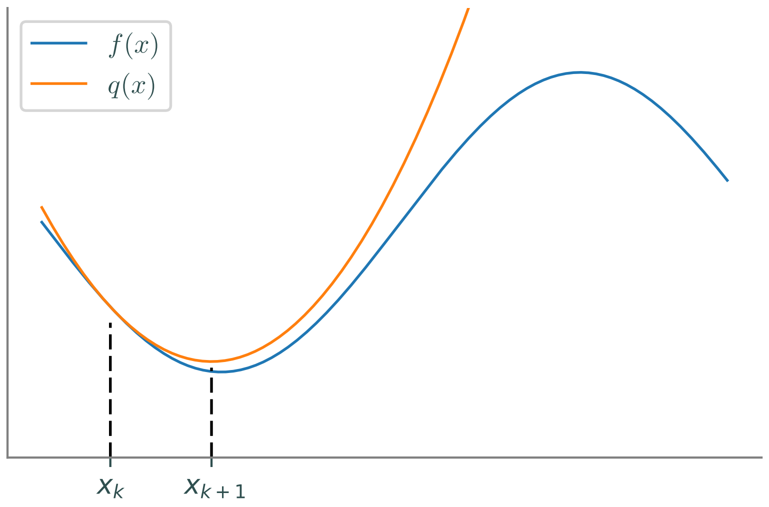

Figure 11.2: A quadratic approximation of f at x_k. The minimizer x_{k+1} of q is close to the minimizer of f.

Newton's method for optimization works well to locate minima when $f''(x) > 0$ on the entire domain. However, it may fail to converge to a minimizer if $f''(x) \leq 0$ for some portion of the domain. If f is not unimodal, the initial guess x_0 must be sufficiently close to a local minimizer x^* in order to converge.

Problem 2. Let $f : \mathbb{R} \to \mathbb{R}$. Write a function that accepts f', f'', a starting point x_0, a stopping tolerance `tol`, and a maximum number of iterations `maxiter`. Implement Newton's method using (11.1) to locate a local optimizer. Return the approximate optimizer, whether or not the algorithm converged, and the number of iterations computed.

Test your function by minimizing $f(x) = x^2 + \sin(5x)$ with an initial guess of $x_0 = 0$. Compare your results to `scipy.optimize.newton()`, which implements the root-finding version of Newton's method.

```
>>> df = lambda x : 2*x + 5*np.cos(5*x)
>>> d2f = lambda x : 2 - 25*np.sin(5*x)
>>> opt.newton(df, x0=0, fprime=d2f, tol=1e-10, maxiter=500)
-1.4473142236328096
```

Note that other initial guesses can yield different minima for this function.

The Secant Method

The second derivative of an objective function is not always known or may be prohibitively expensive to evaluate. The *secant method* solves this problem by numerically approximating the second derivative with a difference quotient.

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

Selecting $x = x_k$ and $h = x_{k-1} - x_k$ gives the following approximation.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \tag{11.2}$$

Inserting (11.2) into (11.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1} f'(x_k) - x_k f'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \tag{11.3}$$

Notice that this recurrence relation requires two previous points (both x_k and x_{k-1}) to calculate the next estimate. This method converges superlinearly—slower than Newton's method, but faster than the golden section search—with convergence criteria similar to Newton's method.

Problem 3. Write a function that accepts a first derivative f', starting points x_0 and x_1, a stopping tolerance `tol`, and a maximum of iterations `maxiter`. Use (11.3) to implement the Secant method. Try to make as few computations as possible by only computing $f'(x_k)$ once for each k. Return the minimizer approximation, whether or not the algorithm converged, and the number of iterations computed.

Test your code with the function $f(x) = x^2 + \sin(x) + \sin(10x)$ and with initial guesses of $x_0 = 0$ and $x_1 = -1$. Plot your answer with the graph of the function. Also compare your results to `scipy.optimize.newton()`; without providing the `fprime` argument, this function uses the secant method. However, it still only takes in one initial condition, so it may converge to a different local minimum than your function.

```
>>> df = lambda x: 2*x + np.cos(x) + 10*np.cos(10*x)
>>> opt.newton(df, x0=0, tol=1e-10, maxiter=500)
-3.2149595174761636
```

Descent Methods

Consider now a function $f : \mathbb{R}^n \to \mathbb{R}$. *Descent methods*, also called *line search methods*, are optimization algorithms that create a convergent sequence $(x_k)_{k=1}^{\infty}$ by the following rule.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{11.4}$$

Here $\alpha_k \in \mathbb{R}$ is called the *step size* and $\mathbf{p}_k \in \mathbb{R}^n$ is called the *search direction*. The choice of $\mathbf{p}_k$ is usually what distinguishes an algorithm; in the one-dimensional case ($n = 1$), $p_k = f'(x_k)/f''(x_k)$ results in Newton's method, and using the approximation in (11.2) results in the secant method.

To be effective, a descent method must also use a good step size α_k. If α_k is too large, the method may repeatedly overstep the minimum; if α_k is too small, the method may converge extremely slowly. See Figure 11.3.

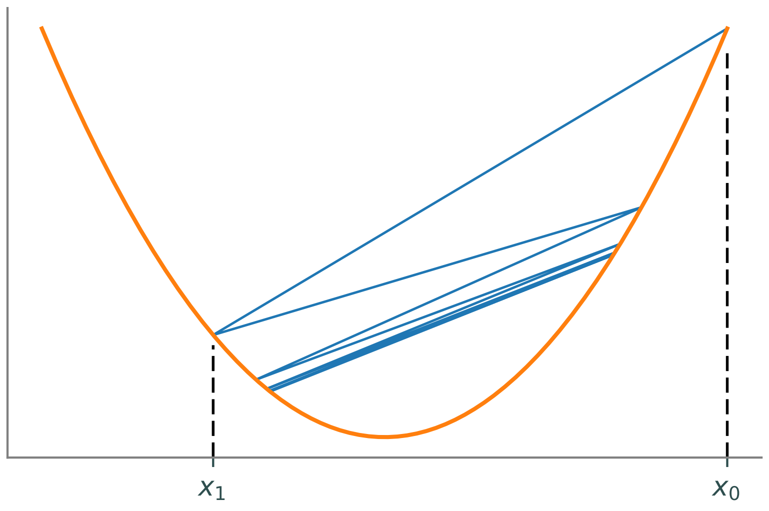

Figure 11.3: If the step size α_k is too large, a descent method may repeatedly overstep the minimizer.

Given a search direction $\mathbf{p}_k$, the best step size α_k minimizes the function $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha\mathbf{p}_k)$. Since f is scalar-valued, $\phi_k : \mathbb{R} \to \mathbb{R}$, so any of the optimization methods discussed previously can be used to minimize ϕ_k. However, computing the best α_k at every iteration is not always practical. Instead, some methods use a cheap routine to compute a step size that may not be optimal, but which is good enough. The most common approach is to find an α_k that satisfies the *Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k\mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1\alpha_k Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k \tag{11.5}$$
$$-Df(\mathbf{x}_k + \alpha_k\mathbf{p}_k)^\mathsf{T}\mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k \tag{11.6}$$

where $0 < c1 < c2 < 1$ (for the best results, choose $c1 << c2$). The condition (11.5) is also called the *Armijo rule* and ensures that the step decreases f. However, this condition is not enough on its own. By Taylor's theorem,

$$f(\mathbf{x}_k + \alpha_k\mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small α_k will always satisfy (11.5) since $Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k < 0$ (as $\mathbf{p}_k$ is a descent direction). The condition (11.6), called the *curvature condition*, ensures that the α_k is large enough for the algorithm to make significant progress.

It is possible to find an α_k that satisfies the Wolfe conditions, but that is far from the minimizer of $\phi_k(\alpha)$. The *strong Wolfe conditions* modify (11.6) to ensure that α_k is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k\mathbf{p}_k)^\mathsf{T}\mathbf{p}_k| \leq c_2|Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (11.6):

$$f(\mathbf{x}_k) + (1-c)\alpha_k Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k\mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha_k Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k,$$

where $0 < c < 1$. These conditions are very similar to the Wolfe conditions (the right inequality is (11.5)), but they do not require the calculation of the directional derivative $Df(\mathbf{x}_k + \alpha_k\mathbf{p}_k)^\mathsf{T}\mathbf{p}_k$.

Backtracking

A *backtracking line search* is a simple strategy for choosing an acceptable step size α_k: start with an fairly large initial step size α, then repeatedly scale it down by a factor ρ until the desired conditions are satisfied. The following algorithm only requires α to satisfy (11.5). This is usually sufficient, but if it finds α's that are too small, the algorithm can be modified to satisfy (11.6) or one of its variants.

Algorithm 11.2 Backtracking using the Armijo Rule

```
1: procedure BACKTRACKING($f$, $Df$, $\mathbf{x}_k$, $\mathbf{p}_k$, $\alpha$, $\rho$, $c$)
2:     Dfp ← $Df(\mathbf{x}_k)^\mathsf{T}\mathbf{p}_k$                ▷ Compute these values only once.
3:     fx ← $f(\mathbf{x}_k)$
4:     while $\big(f(\mathbf{x}_k + \alpha\mathbf{p}_k) > \texttt{fx} + c\alpha\texttt{Dfp}\big)$ do
5:         $\alpha \leftarrow \rho\alpha$
       return $\alpha$
```

Problem 4. Write a function that accepts a function $f : \mathbb{R}^n \to \mathbb{R}$, its derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$, an approximate minimizer $\mathbf{x}_k$, a search direction $\mathbf{p}_k$, an initial step length α, and parameters ρ and c. Implement the backtracking method of Algorithm 11.2. Return the computed step size.

The functions f and Df should both accept 1-D NumPy arrays of length n. For example, if $f(x, y, z) = x^2 + y^2 + z^2$, then f and Df could be defined as follows.

```
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> Df = lambda x: np.array([2*x[0], 2*x[1], 2*x[2]])
```

SciPy's `scipy.optimize.linesearch.scalar_search_armijo()` finds an acceptable step size using the Armijo rule. It may not give the exact answer as your implementation since it decreases α differently, but the answers should be similar.

```
>>> from scipy.optimize import linesearch
>>> from autograd import numpy as anp
>>> from autograd import grad

# Get a step size for f(x,y,z) = x^2 + y^2 + z^2.
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> x = anp.array([150., .03, 40.])          # Current minimizer guesss.
>>> p = anp.array([-.5, -100., -4.5])        # Current search direction.
>>> phi = lambda alpha: f(x + alpha*p)       # Define phi(alpha).
>>> dphi = grad(phi)
>>> alpha, _ = linesearch.scalar_search_armijo(phi, phi(0.), dphi(0.))
```