Lab 16

1-D Optimization

Lab Objective: *Many high-dimensional optimization algorithms rely on one-dimensional optimization methods. In this lab, we implement four* line search *algorithms for optimizing scalar-valued functions defined on* $\mathbb{R}$. *We will generalize some of these approaches to high-dimensional optimization problems in subsequent labs.*

Overview of Line Search Algorithms

Imagine you are hiking a steep mountain. When it is time to head home, thick fog gathers around, reducing visibility to just a couple of feet. How can you find your way back with such limited visibility? One strategy is to pick a downhill direction and follow that direction as far as you can, or until it starts leading upward again. Then choose another downhill direction, and take that as far as you can, repeating the process. By always choosing a downhill direction, you hope to eventually make it back to the base of the mountain.

This is the basic approach of *line search algorithms* for numerical optimization. Let f be a scalar-valued function. The goal is to find the *global minimizer* $\mathbf{x}^*$ such that $f(\mathbf{x}^*) \leq f(\mathbf{x})$ for all $\mathbf{x}$ in the domain of f. After choosing an initial guess $\mathbf{x}_0$, we produce a sequence of approximations $\mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{x}_3,\ \ldots$ using the rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k. \tag{16.1}$$

The point $\mathbf{p}_k$ is the *search direction* (which way to go) and the scalar α_k is called the *step size* (how far to go). The choice of search direction and step size at each step is what defines different line search algorithms.

Derivative versus Derivative-Free Methods

A function's derivative provides information about how the value of the function changes at each point, and can be used to determine local optima. Unfortunately, not all objective functions are differentiable, and many are difficult or costly to differentiate. Thus some line search methods utilize the derivative of the objective function, but others do not.

Interval Approximation Methods for Unimodal Functions

A function $f : [a, b] \to \mathbb{R}$ satisfies the *unimodal property* if it has just **one** local (hence global) minimum and is monotonic to the left and right of the minimum. The following line search methods optimize a unimodal function by partitioning the function's domain into progressively smaller intervals that contain the minimizer.

Golden Section Search

Let $f : [a, b] \to \mathbb{R}$ be unimodal. Starting with the closed interval $[a, b]$, the *golden section search* finds a smaller interval that contains x^* as follows.

Choose two points a' and b' with $a' < b'$. If $f(a') \geq f(b')$, the unimodal property guarantees that the minimizer must be in the interval $[a', b]$, for otherwise the function f would have a local minimum in both $[a, a']$ and $[a', b]$. In the next step, we repeat the process over the interval $[a', b]$. If instead $f(a') \leq f(b')$, then we choose the interval $[a, b']$ for the next step. Finally, if $f(a') = f(b')$, then it does not matter which interval is chosen.

It turns out that there is an optimal choice for the two test points a' and b'. Given the current interval $[a, b]$, choose a' and b' satisfying

$$\begin{aligned} a' &= a + \rho(b - a) \\ b' &= a + (1 - \rho)(b - a), \end{aligned}$$

where $\rho = \frac{1}{2}(3 - \sqrt{5}) \approx 0.382$ (this constant is related to the famous *golden ratio*, hence the name of the algorithm).

At each step, the interval is reduced by a factor of $1 - \rho$, which means that after n steps, the minimizer is pinned down to within an interval approximately $(0.618)^n$ times the length of the original interval. Note that this rate of convergence is independent of the objective function.

Problem 1. Write a function that accepts a callable function f, interval limits a and b, and a number of iterations `niter` to calculate. Implement the golden section search as described above, returning the midpoint of the final interval.

Use your function to minimize $f(x) = e^x - 4x$ on the interval $[0, 3]$. How many steps do you need to take to get within .001 of the true minimizer?

Bisection Algorithm

This method is similar to the Golden Ratio method, but instead of cutting our interval into two overlapping sections, we divide the interval evenly in half, and then use the derivative $f'(x)$ to determine where the minimizer lies. This method takes advantage of the fact that a unimodal function's derivative is strictly less than 0 to the left of the minimizer, and strictly greater than 0 to the right.

For a unimodal function $f : [a, b] \to \mathbb{R}$, take the midpoint, $x_1 = \frac{b+a}{2}$. If $f'(x_1) > 0$, the critical point must lie in the interval $[a, x_1]$; otherwise, the critical point lies in the other half of the interval, $[x_1, b]$. We repeat this process on the correct interval until the desired accuracy is achieved.

At each step, the interval is reduced by a factor of two. This is slightly quicker than the factor of $1-\rho$ in the Golden Section search method. Like the golden section search, the convergence of this method is independent of the objective function.

Problem 2. Write a function that accepts a callable function f, interval limits a and b, and a number of iterations `niter` to calculate. Implement the bisection algorithm as described above, returning the midpoint of the final interval.

Use your function to minimize the same objective function as in Problem 1. How many steps do you need to take to get within .001 of the true minimizer? Time both algorithms and report which one is faster to converge.

Newton's Method and Quasi-Newton Methods

Newton's Method, is a basic line search algorithm that uses the derivatives of the function to select a direction and step size.

To use this method, we need a function that is twice differentiable. The idea is to approximate the function with a quadratic polynomial and then solve the trivial problem of minimizing the polynomial. Doing so in an iterative manner can lead us to the actual minimizer. Let f be a function satisfying the appropriate conditions, and let us make an initial guess, x_0. The relevant quadratic approximation to f at x_0 is

$$q(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2,$$

which is the second-degree Taylor polynomial for f centered at x_0. The minimum for this quadratic function is easily found by solving $q'(x) = 0$, and we take the obtained x-value as our new approximation. The formula for the $(k+1)$-th approximation, which the reader can verify, is

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

In the one dimensional case, there are only two search directions: to the right $(+)$ or to the left $(-)$. Newton's method chooses the search direction $\mathbf{p}_k = \text{sign}(-f'(x_k)/f''(x_k))$ and the step size $\alpha_k = |f'(x_k)/f''(x_k)|$.

The convergence properties of this sequence depend heavily on the initial guess x_0 and the function f. Roughly speaking, if x_0 is sufficiently close to the actual minimizer, and if f is well-approximated by parabolas, then one can expect the sequence to converge quickly. However, there are cases when the sequence converges slowly or not at all. See Figure 16.1.

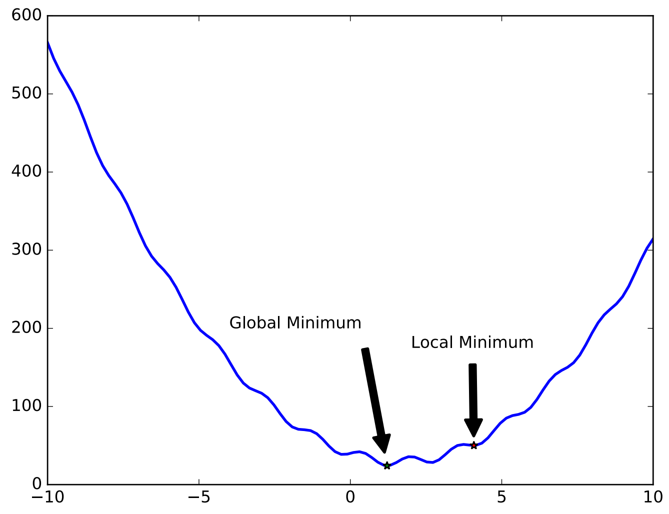

Figure 16.1: The results of Newton's Method using two different initial guesses. The global minimizer was correctly found with initial guess of 1. However, an initial guess of 4 led to only a local minimum.

Problem 3. Implement Newton's Method. You will write a function that takes a function and its two derivatives, as well as a starting point. It will return the minimizer.

Use this function to minimize $f(x) = x^2 + \sin(5x)$ with an initial guess of $x_0 = 0$. Now try other initial guesses farther away from the true minimizer, and note when the method fails to obtain the correct answer.

One-Dimensional Secant Method

Sometimes we would like to use Newton's Method, but for whatever reason we don't have a second derivative. Maybe calculating it is too costly or the function is not twice differentiable. In this situation we can approximate the second derivative using the first derivative. The approximation using a secant calculation, hence the name. We use the same basic algorithm as in Newton's Method, but with an approximation for the second derivative of the objective function.

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''_{approx}(x_n)},$$

where we approximate the second derivative as follows:

$$f''_{approx}(x_n) = \frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}.$$

This gives us the final equation:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f'(x_n) - f'(x_{n-1})} f'(x_n),$$

Notice that this equation requires two initial points (both x_n and x_{n-1}) to calculate the next estimate.

Problem 4. Write a function that accepts a scalar-valued function f, its first derivative Df, and two starting points. Use the secant method to find and return the minimizer of f.

Use your function to minimize $f(x) = x^2 + \sin(x) + \sin(10x)$ with initial guesses of $x_0 = 0$ and $x_1 = -1$. Now try other initial guesses farther away from the true minimizer, and note when the method fails to obtain the correct answer.

(Hint: You may want to plot this function to understand why this problem is so sensitive to the starting point.)

General Line Search Methods

Step Size Calculation

Given a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ that we wish to minimize, and assuming that we already have a current point $\mathbf{x}_k$ and direction $\mathbf{p}_k$ in which to search, how do we choose our step size α_k? If our step size is too small, we will not make good progress toward the minimizer, and convergence will be slow. If the step size is too large, however, we may overshoot and produce points that are far away from the solution. A common approach to pick an appropriate step size involves the *Wolfe conditions*:

$$\begin{aligned} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) &\leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k, && (0 < c_1 < 1), \\ \nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\mathsf{T} \mathbf{p}_k &\geq c_2 \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k, && (c_1 < c_2 < 1). \end{aligned}$$

The search direction p_k is often required to satisfy $p_k^\mathsf{T} \nabla f(\mathbf{x}_k) < 0$, in which case it is called a *descent direction*, since the function is guaranteed to decrease in this direction. Generally speaking, choosing a step size α_k satisfying these conditions ensures that we achieve sufficient decrease in the function and also that we do not terminate the search at a point of steep decrease (since then we could achieve even better results by choosing a slightly larger step size). The first condition is known as the *Armijo* condition.

We will discuss methods of finding search directions in future labs. For now, we will discuss one simple algorithm for finding an appropriate step size which satisfies the Armijo conditions.

Backtracking

Finding such a step size satisfying these conditions is not always an easy task, however. One simple approach, known as *backtracking*, starts with an initial step size α, and repeatedly scales it down until the Armijo condition is satisfied. That is, choose $\alpha > 0, \rho \in (0,1), c \in (0,1)$, and while

$$f(\mathbf{x}_k + \alpha \mathbf{p}_k) > f(\mathbf{x}_k) + c\alpha \nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k,$$

re-scale $\alpha = \rho\alpha$. Once the loop terminates, set $\alpha_k = \alpha$. Note that the value $\nabla f(\mathbf{x}_k)^\mathsf{T} \mathbf{p}_k$ remains fixed for the duration of the backtracking algorithm, and hence need only be calculated once at the beginning.

Problem 5. Implement this backtracking algorithm using the function definition in the spec file. Your function will accept a function, its derivative, a starting point, and the direction p.

Line Search in SciPy

SciPy's `optimize` module contains implementations of various optimization algorithms, including several line search methods. In particular, the module provides a useful routine for calculating a step size satisfying the Wolfe Conditions described above, which is more robust and efficient than our simple backtracking approach. We recommend its use for the remainder of this lab. The function is called `line_search()`, and accepts several arguments. We can typically leave the keyword arguments at their default values, but we do need to pass in the objective function, its gradient, the current point, and the search direction. The following code gives an example of its usage, using the objective function $f(x, y) = x^2 + 4y^2$.

```
>>> import numpy as np
>>> from scipy.optimize import line_search
>>>
>>> def objective(x):
>>>     return x[0]**2 + 4*x[1]**2
>>>
>>> def grad(x):
>>>     return np.array([2*x[0], 8*x[1]])
>>>
>>> x = np.array([1., 3.]) # current point
>>> p = -grad(x)           # current search direction
>>> a = line_search(objective, grad, x, p)[0]
>>> print a
0.125649913345
```

Note that the function returns a tuple of values, the first of which is the step size.