Lab 1

Line Search Algorithms

Lab Objective: *Investigate various Line-Search algorithms for numerical optimization.*

Overview of Line Search Algorithms

Imagine you are out hiking on a mountain, and you lose track of the trail. Thick fog gathers around, reducing visibility to just a couple of feet. You decide it is time to head back home, which is located in the valley located near the base of the mountain. How can you find your way back with such limited visibility? The obvious way might be to pick a direction that leads downhill, and follow that direction as far as you can, or until it starts leading upward again. Then you might 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 bottom of the valley, where you live.

This is the basic approach of line search algorithms for numerical optimization. Suppose we have a real-valued function f that we wish to minimize. Our goal is to find the point x^* in the domain of f such that $f(x^*)$ is the smallest value in the range of f. For some functions, we can use techniques from calculus to analytically obtain this minimizer. However, in practical applications, this is often impossible, especially when we need a system that works for a wide class of functions. A line search algorithm starts with an initial guess at the minimizer, call it x_0, and iteratively produces a sequence of points $x_1, x_2, x_3, \ldots$ that hopefully converge to the minimizer x^*. The basic iteration to move from x_k to x_{k+1} involves two steps: first, choosing a search direction p_k in which to proceed from the current point, and second, specifying a step size α_k to travel in this direction. The next point is determined by the formula

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

This procedure is called a line search because at each iteration, we are simply examining the function in a particular linear direction. The choice of the step size α_k is often chosen by solving a one-dimensional optimization problem in the given

direction. In this lab, we will discuss approaches to choosing the step size and the search direction.

One-Dimensional Newton's Method

Let us first start out with a basic task: minimizing a function of one variable. We will use a popular approach known as Newton's Method, which 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 real-valued function of a real variable 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 is

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

or just 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 $(n+1)$-th approximation, which the reader can verify, is

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

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 $\text{sign}(-f'(x_n)/f''(x_n))$ and the step size $|f'(x_n)/f''(x_n)|$.

As is typical with optimization algorithms, Newton's Method generates a sequence of points or successive approximations to the minimizer. However, 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 1.1.

Problem 1. Implement Newton's Method as described using the following function declaration.

```
def newton1d(f, df, ddf, x, niter=10):
    '''
    Perform Newton's method to minimize a function from R to R.

    Parameters
    ----------
    f : callable function object
        The objective function (twice differentiable)
    df : callable function object
        The first derivative
    ddf : callable function object
```

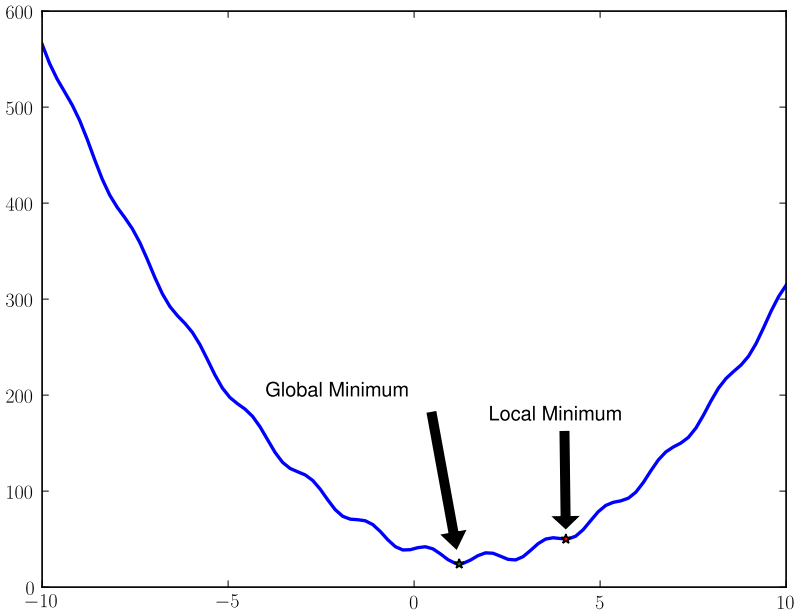

Figure 1.1: The results of Newton's Method using two different initial guess. The global minimizer was correctly found with initial guess of 1. However, an initial guess of 4 led to only a local minimum.

```
        The second derivative
    x : float
        The initial guess
    niter : integer
        The number of iterations

    Returns
    ------
        min : float
            The approximated minimizer
    '''
    pass
```

Use this function to minimize $x^2 + \sin(5x)$ 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.

General Line Search Methods

Step Size Calculation

We now examine Line Search methods in more generality. 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 x_k and direction 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*:

$$\begin{aligned} f(x_k + \alpha_k p_k) &\leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k, && (0 < c_1 < 1), \\ \nabla f(x_k + \alpha_k p_k)^T p_k &\geq c_2 \nabla f_k^T p_k, && (c_1 < c_2 < 1). \end{aligned}$$

Here, we use the shorthand notation ∇f_k to mean the gradient of f evaluated at the point x_k. The search direction p_k is often required to satisfy $p_k^T \nabla f_k < 0$, 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.

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

$$f(x_k + \alpha p_k) > f(x_k) + c\alpha \nabla f_k^T p_k,$$

re-scale $\alpha := \rho\alpha$. Once the loop terminates, set $\alpha_k = \alpha$. Note that the value $\nabla f_k^T p_k$ remains fixed for the duration of the backtracking algorithm, and hence need only be calculated once at the beginning.

Problem 2. Implement this backtracking algorithm using the following function declaration.

```
def backtracking(f, slope, x, p, a=1, rho=.9, c=10e-4):
    '''
    Perform a backtracking line search to satisfy the Armijo condition.

    Parameters
    ----------
    f : callable function object
        The objective function
    slope : float
        The value of grad(f)^T p
    x : ndarray of shape (n,)
        The current iterate
    p : ndarray of shape (n,)
```

```
        The current search direction
    a : float
        The intial step length (set to 1 in Newton and quasi-Newton ↩
            methods)
    rho : float
        A number in (0,1)
    c : float
        A number in (0,1)

    Returns
    -------
        alpha : float
            The computed step size satisfying the Armijo condition.
    '''
    pass
```

Choosing a Search Direction

There are many different ways to choose a search direction p_k. As noted earlier, it is usually a requirement to choose a descent direction. We will compare two methods, both using derivative information about the function.

Gradient Descent. Recall that the gradient of a function at a given point gives the direction in which the function is increasing fastest. Thus, the negative of the gradient points in the direction of fastest decrease. In the method of Gradient Descent, we choose our search direction to be this direction of steepest descent, that is,

$$p_k = -\nabla f_k.$$

This is a very natural choice, since we seem to be approaching the minimum value as fast as possible. However, depending on the nature of the objective function, convergence may be slow. See Figure 1.2.

Newton's Method. We now generalize the one-dimensional Newton's method presented above. We use both the gradient and the Hessian matrix (which gives information on the curvature of the function at a given point) to choose a search direction. This is more computationally intensive, but it leads to very fast convergence in many cases. See Figure 1.2. Our search direction is

$$p_k = -\nabla^2 f_k^{-1} \nabla f_k,$$

where $\nabla^2 f_k^{-1}$ is the inverse of the Hessian matrix of f at the point x_k. In other words, p_k is the solution to the linear system

$$\nabla^2 f_k p_k = -\nabla f_k.$$

Problem 3. Implement the Gradient Descent algorithm and Newton's Method using the following function declarations. In each function, you should call your backtracking function with values $\alpha = 1, \rho = .9$, and $c = 10^{-4}$. The

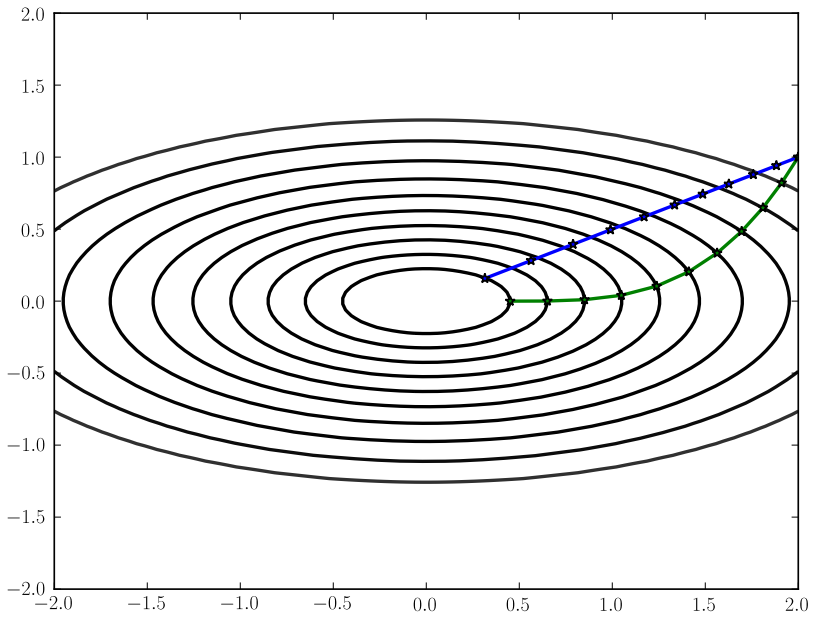

Figure 1.2: Paths generated by Gradient Descent (green) and Newton's Method (blue). Note that the Newton path takes a more direct route toward the minimizer (located at the origin).

`scipy.linalg` module may be useful when computing the search direction in Newton's Method.

```
def gradientDescent(f, df, x, niter=10):
    '''
    Minimize a function using gradient descent.

    Parameters
    ----------
    f : callable function object
        A differentiable real-valued function
    df : callable function object
        The gradient of the function
    x : ndarray of shape (n,)
        The initial point
    niter : integer
        The number of iterations to run.

    Returns
    -------
        pts: list of ndarrays
            The sequence of points generated
    '''
    pass
```

```
def newtonsMethod(f, df, ddf, x, niter=10):
    '''
    Minimize a function using Newton's method.

    Parameters
    ----------
    f : callable function object
        Real-valued, twice-differentiable function
    df : callable function object
        The gradient of the function
    ddf : callable function object
        The Hessian of the function
    x : ndarray of shape (n,)
        The initial point
    niter : integer
        The number of iterations

    Returns
    -------
    pts : list of ndarrays
        The sequence of points generated
    '''
    pass
```

Line Search in SciPy

The SciPy module `scipy.optimize` 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 2*x*np.array([1, 4])
>>>
>>> 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. We have illustrated the very basic use of this function. See the documentation for further uses.

Non-linear Least Squares Problems

We now discuss a very important class of problems known as Least Squares problems. These are unconstrained optimization problems that seek to minimize an objective function of the form

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x),$$

where each $r_i : \mathbb{R}^n \to \mathbb{R}$ is smooth, and $m \geq n$. Such problems arise in many scientific fields, including economics, physics, and statistics. Linear Least Squares problems form an important subclass, and can be solved directly without the need for an iterative method. At present we will focus on the non-linear case, which can be solved with a line search method.

To motivate the problem further, suppose you are given a set of data points, and you have some kind of model for the data. You need to choose particular values for the parameters in your model, and you wish to do so in a way that "best fits" the observed data. What do we mean by "best fit"? We need some way to measure the error between our model and the data set, and then minimize this error. The best fit will correspond to the choice of parameters that minimize the error function.

More formally, suppose we are given the data points $(t_1, y_1), (t_2, y_2), \ldots, (t_m, y_m)$, where $y_i \in \mathbb{R}$ and $t_i \in \mathbb{R}^n$ for $i = 1, \ldots, m$. Let $\phi(x, t)$ be our model for this data set, where x is a vector of parameters of the model, and $t \in \mathbb{R}^n$. We can measure the error at the i-th data point by the value

$$r_i(x) := \phi(x, t_i) - y_i,$$

and by summing the squares of these errors, we obtain our non-linear least squares objective function:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x).$$

The individual functions r_i that measure the error between the model and the data point are known as *residuals*, and we can aggregate these functions into a *residual vector*

$$r(x) := (r_1(x), r_2(x), \ldots, r_m(x))^T.$$

The Jacobian of $r(x)$ can be expressed in terms of the gradients of each r_i as follows:

$$J(x) = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_m(x)^T \end{bmatrix}$$

You can further verify that

$$\nabla f(x) = J(x)^T r(x),$$
$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x).$$

That second term in the formula for $\nabla^2 f$ involves second derivatives and can be problematic to compute. Often in practice, this term is small, either because the residuals themselves are small, or are nearly affine in a neighborhood of the solution and hence the second derivatives are small. The simplest method for solving the nonlinear least squares problem, known as the *Gauss-Newton Method*, exploits this observation, simply ignoring the second term and making the approximation

$$\nabla^2 f(x) \approx J(x)^T J(x).$$

The method then proceeds in a manner similar to Newton's Method. In particular, at the k-th iteration, we choose a search direction p_k that solves the linear system

$$J_k^T J_k p_k = -J_k^T r_k.$$

For convenience, we summarize these steps in Algorithm 1.1.

Algorithm 1.1 Gauss-Newton Method

procedure GAUSS-NEWTON
  Choose initial parameter vector x_0
  $k \leftarrow 0$
  while $J_k^T r_k \neq 0$ **do**
    solve $J_k^T J_k p_k = -J_k^T r_k$
    choose step size α_k satisfying Wolfe Conditions.
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $k \leftarrow k + 1$

Problem 4. Implement the Gauss-Newton method using the following function declaration.

```
def gaussNewton(f, df, jac, r, x, niter=10):
    '''
    Solve a nonlinear least squares problem with Gauss-Newton method.

    Parameters
    ----------
    f : callable function object
        The objective function
    df : callable function object
        The gradient of f
    jac : callable function object
        The jacobian of residual vector
    r : callable function object
        The residual vector
    x : ndarray of shape (n,)
        The initial point
```

```
    niter : integer
        The number of iterations

    Returns
    -------
    min : ndarray of shape (n,)
        The minimizer
    '''
    pass
```

Feel free to use SciPy functions to solve linear systems and calculate step sizes in your algorithm.

Let us work through an example of a nonlinear least squares problem. Suppose we have data points generated from a sine function and slightly perturbed by gaussian noise. In Python we can generate such data as follows:

```
>>> t = np.arange(10)
>>> y = 3*np.sin(0.5*t)+ 0.5*np.random.randn(10)
```

Now we write Python functions for our model, the residual vector, the Jacobian, the objective function, and the gradient. The calculations for all of these are straight forward.

```
>>> def model(x, t):
>>>     return x[0]*np.sin(x[1]*t)
>>> def residual(x):
>>>     return model(x, t) - y
>>> def jac(x):
>>>     ans = np.empty((10,2))
>>>     ans[:,0] = np.sin(x[1]*t)
>>>     ans[:,1] = x[0]*t*np.cos(x[1]*t)
>>>     return ans
>>> def objective(x):
>>>     return .5*(residual(x)**2).sum()
>>> def grad(x):
>>>     return jac(x).T.dot(residual(x))
```

By inspecting our data, we might make an initial guess for the parameters $x_0 = (2.5, 0.6)$. We are now ready to use our `gaussNewton` function to find the least squares solution.

```
>>> x0 = np.array([2.5,.6])
>>> x = gaussNewton(objective, grad, jac, residual, x0, niter=10)
```

We can plot everything together to compare our fitted model with the data and the original sine curve from which the data were generated.

```
dom = np.linspace(0,10,100)
plt.plot(t, y, '*')
plt.plot(dom, 3*np.sin(.5*dom), '--')
plt.plot(dom, x[0]*np.sin(x[1]*dom))
plt.show()
```

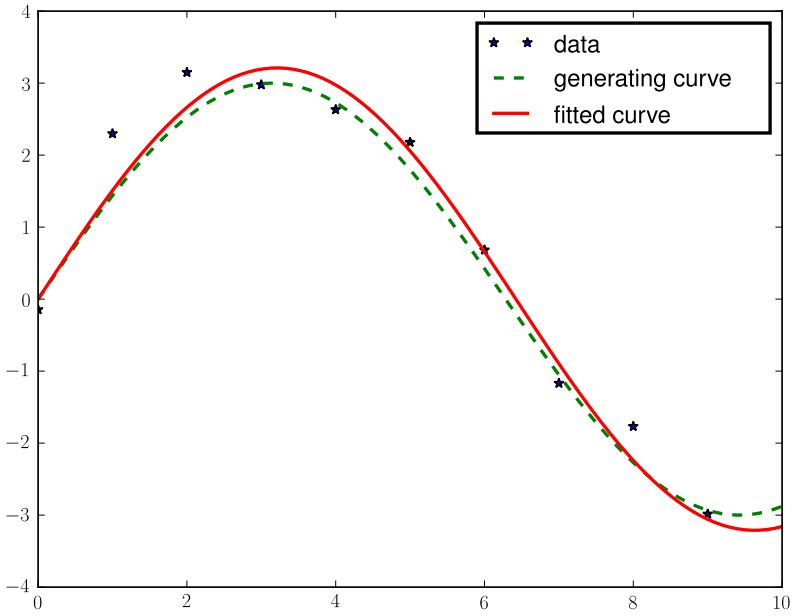

Figure 1.3: Perturbed data (stars) generated from a sine curve (dashed line), together with the fitted sine curve (solid line).

The results are shown in Figure 1.3. As you can see, after just 10 iterations, we have found a very good fit.

Non-linear Least Squares in Python

The module `scipy.optimize` also has a method to solve non-linear least squares problem, and it is quite convenient. The function is called `leastsq`, and in its most basic use, you only need to pass in the residual function and starting point as arguments. In the example above, we simply need to execute the following code:

```
>>> from scipy.optimize import leastsq
>>> x2 = leastsq(residual, x0)[0]
```

This should give us the same answer, but much faster.

Problem 5. We have census data giving the population of the United States every ten years since 1790. For convenience, we have entered the data in Python below, so that you may simply copy and paste.

```
>>> #Start with the first 8 decades of data
>>> years1 = np.arange(8)
>>> pop1 = np.array([3.929, 5.308, 7.240, 9.638, 12.866,
```

```
>>>                 17.069, 23.192, 31.443])
>>>
>>> #Now consider the first 16 decades
>>> years2 = np.arange(16)
>>> pop2 = np.array([3.929, 5.308, 7.240, 9.638, 12.866,
>>>                 17.069, 23.192, 31.443, 38.558, 50.156,
>>>                 62.948, 75.996, 91.972, 105.711, 122.775,
>>>                 131.669])
```

Consider just the first 8 decades of population data. By plotting the data and having an inclination that population growth tends to be exponential, it is reasonable to hypothesize an exponential model for the population, that is,

$$\phi(x_1, x_2, x_3, t) = x_1 \exp(x_2(t + x_3)).$$

By inspection, find a reasonable initial guess for the parameters (x_1, x_2, x_3) (i.e. $(150, .4, 2.5)$). Write a function for this model in Python, along with the corresponding residual vector, and fit the model using the `leastsq` function. Plot the data against the fitted curve, to see how close you are.

Now consider all 16 decades of data. If you plot your curve from above with this more complete data, you will see that the model is no longer a good fit. Instead, the data suggest a logistic model, which also arises from a differential equations treatment of population growth. Thus, your new model is

$$\phi(x_1, x_2, x_3, t) = \frac{x_1}{1 + \exp(-x_2(t + x_3))}.$$

By inspection, find a reasonable initial guess for the parameters (x_1, x_2, x_3) (i.e. $(150, .4, -15)$). Again, write Python functions for the model and the corresponding residual vector, and fit the model. Plot the data against the fitted curve. It should be a good fit.