14 Newton and Quasi-Newton Methods

Lab Objective: *Newton's method is generally useful because of its fast convergence properties. However, Newton's method requires the explicit calculation of the second derivative (i.e., the Hessian matrix) at each step, which is computationally costly. Quasi-Newton methods modify Newton's method so that the Hessian does not have to be computed at each step, thus making computations faster. This generally comes at the cost of slower convergence speed, but the increased computation speed can make these methods more effective in many cases.*

Newton's Method

At this point, we have discussed Newton's Method several times. In the n dimensional version, the next step is given by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - D^2 f(\mathbf{x}_k)^{-1} Df(\mathbf{x}_k)^T \tag{14.1}$$

Problem 1. Write code implementing Newton's method in n Dimensions. Ensure that it takes its Jacobian, and Hessian as arguments and returns the minimizer. Test it on the Rosenbrock function:

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

using two different starting points $(-2, 2)$ and $(10, -10)$.

Broyden's Method

One quasi-Newton method is known as Broyden's method. Broyden's Method is a multidimensional version of the secant method we have discussed previously. Just like the secant method approximates the second derivative of a function by using the first derivatives at nearby points, Broyden's Method uses the Jacobian to update an initial Hessian matrix.

Broyden's Method in n Dimensions

If we have the point $\mathbf{x}_k$ and the Hessian $D^2 f(\mathbf{x}_k)$ at that point, we can use the following equation to select our guess for the zero of the function:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - D^2 f(\mathbf{x}_k)^{-1} Df(\mathbf{x}_k)^T. \tag{14.2}$$

This is precisely Newton's method. However, since calculating the Hessian at each step is costly, we instead estimate the Hessian (just as we used a secant line to approximate the derivative in the one-dimensional case).

The idea is to construct the best rank-one approximation of the Hessian at each iteration. This approximation, denoted A_{k+1}, must satisfy

$$Df(\mathbf{x}_{k+1}) - Df(\mathbf{x}_k) = (\mathbf{x}_{k+1} - \mathbf{x}_k)^T A_{k+1}. \tag{14.3}$$

In multiple dimensions, this equation will be underdetermined (i.e., many A_{k+1}'s will satisfy the equation). Suppose that we have a previous estimate of the Hessian A_k at the point $\mathbf{x}_k$ (note that for the first iteration, we plug in the starting point to the Hessian as our first approximation). We can then find the best approximation of A_{k+1} that minimizes $\|A_{k+1} - A_k\|$. If we define $\mathbf{y}_k = Df(\mathbf{x}_{k+1})^T - Df(\mathbf{x}_k)^T$ and $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, this requirement is uniquely fulfilled by the following:

$$A_{k+1} = A_k + \frac{\mathbf{y}_k - A_k \mathbf{s}_k}{\|\mathbf{s}_k\|^2} \mathbf{s}_k^T. \tag{14.4}$$

After we have obtained the approximation of the Hessian (Equation 14.4), we can apply Equation 14.2 to find $\mathbf{x}_{k+1}$. We can then repeat this process until we have (presumably) converged to a zero of the function.

Problem 2. Write code implementing Broyden's method. Test it on the function:

$$f(x, y) = e^{x-1} + e^{1-y} + (x - y)^2$$

using starting points $(2, 3)$ and $(3, 2)$.

Note

We can often make Broyden's method faster using the Sherman-Morrison-Woodbury Formula. This formula allows us to efficiently calculate the inverse of a matrix when we add a low rank update to that matrix. After manipulation of the Sherman-Morrison-Woodbury Formula, we obtain the following:

$$A_{k+1}^{-1} = A_k^{-1} + \frac{\mathbf{s}_k - A_k^{-1}\mathbf{y}_k}{\mathbf{s}_k^T A_k^{-1} \mathbf{y}_k} (\mathbf{s}_k^T A_k^{-1})$$

Thus, we can calculate the inverse of the Hessian in the first step of our algorithm and then calculate an update to the inverse at each step using the above formula.

BFGS

To be a descent method, we need a monotonically decreasing sequence of functions. In other words, $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. However, if the Hessian or the approximated Hessian is not positive definite, we cannot expect that $f(\mathbf{x}_k) < f(\mathbf{x}_{k-1})$. A drawback of Broyden's method is that the Hessian approximations are not always positive definite, even if $D^2 f(\mathbf{x}_k) > 0$. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is an adjustment to Broyden's method that maintains a positive-definite Hessian approximation. To do this, it makes a rank-two approximation instead of a rank-one approximation. It uses the same update of $\mathbf{x}_k$ as Broyden's method, but with a different update of A_k:

$$A_{k+1} = A_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{A_k \mathbf{s}_k \mathbf{s}_k^T A_k}{\mathbf{s}_k^T A_k \mathbf{s}_k} \tag{14.5}$$

where $\mathbf{y}_k$ and $\mathbf{s}_k$ have the same definitions as given above.

Problem 3. Implement the BFGS method in a new function. Your function should take in the Jacobian, the Hessian, a starting point, a number of iterations, and a stopping tolerance, and return the approximated root along with the number of iterations it took. Test your implementation with the same function given in Problem 2. Experiment with different starting points. Is BFGS faster than Broyden's method? Are there cases (i.e., different starting points or varying number of iterations) where one implementation is better than the other?

Comparing Newton and Quasi-Newton Methods

Different optimization algorithms are more efficient in different situations. If the Jacobian and Hessian are readily available and the Hessian is easily inverted, the standard Newton's Method is probably the best option. If the Hessian is not available or difficult to compute, then using Broyden's Method may be a better option. In circumstances where computing the inverse of the Hessian is difficult, BFGS will allow us to update the inverted Hessian at each step without repeatedly inverting a matrix.

Problem 4. Compare the performance of Newton, Broyden, and BFGS on the following functions:

$$f(x, y) = 0.26(x^2 + y^2) - 0.48xy$$

$$f(x, y) = sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

Output the number of iterations of each algorithm as well as the total time each algorithm takes to run. Calculate the time per iteration for each algorithm and compare. You should see that Newton's Method takes more time per iteration, but that is takes fewer steps than the other algorithms.

The Gauss-Newton Method

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

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

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 $y_i, t_i \in \mathbb{R}$ for $i = 1, \ldots, m$. Let $\phi(\mathbf{x}, \mathbf{t})$ be our model for this data set, where $\mathbf{x}$ is a vector of parameters of the model, and $\mathbf{t} \in \mathbb{R}^n$. We can measure the error at the i-th data point by the value

$$r_i(\mathbf{x}) := \phi(x_i, t_i) - y_i,$$

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

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

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*

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

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

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

You can further verify that

$$\nabla f(\mathbf{x}) = J(\mathbf{x})^T r(\mathbf{x}),$$

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

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

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

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

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

Thus, at each iteration, we find $\mathbf{x}_{k+1}$ as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (J(\mathbf{x}_k)^T J(\mathbf{x}_k))^{-1} J(\mathbf{x}_k)^T \mathbf{r}(\mathbf{x}_k). \tag{14.6}$$

Problem 5. Implement the Gauss-Newton Method. Your function should take the gradient of the objective function Df, the residual vector r, and the Jacobian of the residual vector J. Return the minimizer.

Test your function by using `optimize.leastsq` and comparing your minimizer with Scipy's minimizer. Use the Jacobian function, residual function, and starting point given in the example below.

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(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()
```

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 6. 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.