12 Newton and Quasi-Newton Methods

Lab Objective: *Newton's method is the basis of several iterative methods for optimization. Though it converges quickly, it is often very computationally expensive. Variants on Newton's method, including BFGS, remedy the problem somewhat by numerically approximating Hessian matrices. In this lab we implement Newton's method, BFGS, and the Gauss-Newton method for nonlinear least squares problems.*

Newton's Method

For $g : \mathbb{R} \to \mathbb{R}$, Newton's method finds a root $\bar{x}$ of the equation $g(x) = 0$ with the following rule.

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)} \tag{12.1}$$

Let $f : \mathbb{R} \to \mathbb{R}$. Substituting $g = f'$ into (12.1) yields an iterative method for locating a critical point x^* of f satisfying $f'(x^*) = 0$.

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{12.2}$$

This technique generalizes to higher dimensions. For $g : \mathbb{R}^n \to \mathbb{R}^n$, the following iterative technique finds $\bar{\mathbf{x}}$ such that $g(\bar{\mathbf{x}}) = \mathbf{0}$.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - Dg(\mathbf{x}_k)^{-1} g(\mathbf{x}_k) \tag{12.3}$$

Now let $f : \mathbb{R}^n \to \mathbb{R}$. To calculate an optimal value $\mathbf{x}^*$ of f satisfying $Df(\mathbf{x}^*) = \mathbf{0}$, plug $g = Df$ into (12.3) to get the following equation.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - D^2 f(\mathbf{x}_k)^{-1} Df(\mathbf{x}_k)^\mathsf{T} \tag{12.4}$$

Here the first derivative $Df : \mathbb{R}^n \to \mathbb{R}^n$ evaluates to the row vector $Df(\mathbf{x}) = [D_1 f(\mathbf{x}) \ \dots \ D_n f(\mathbf{x})]$, and the second derivative $D^2 f : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ evaluates to the $n \times n$ *Hessian* matrix

$$D^2 f(\mathbf{x}) = \begin{bmatrix} D_1 D_1 f(\mathbf{x}) & \dots & D_n D_1 f(\mathbf{x}) \\ D_1 D_2 f(\mathbf{x}) & \dots & D_n D_2 f(\mathbf{x}) \\ \vdots & & \vdots \\ D_1 D_n f(\mathbf{x}) & \dots & D_n D_n f(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}.$$

Problem 1. Write a function that accepts functions $Df : \mathbb{R}^n \to \mathbb{R}^n$ and $D^2f : \mathbb{R}^n \to \mathbb{R}^{n\times n}$, a starting point $\mathbf{x}_0 \in \mathbb{R}^n$, an integer `maxiter`, and a stopping tolerance `tol`. Use Newton's method in (12.4) to optimize f. Return the final estimate $\mathbf{x}_k$, whether or not the method converged (`True` or `False`), and the number of iterations computed.

Your implementation should include the following items.

- Iterate until either $\|Df(\mathbf{x}_k)\|_\infty <$ `tol` or $k >$ `maxiter`. The criteria $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| <$ `tol` is also common, but making sure Df is near zero works better in many circumstances.
- Instead of inverting $D^2f(\mathbf{x}_k)$ at each step, solve the equation $D^2f(\mathbf{x}_k)\mathbf{z}_k = Df(\mathbf{x}_k)^\mathsf{T}$ and compute $\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{z}_k$. In other words, use `la.solve()` instead of `la.inv()`.
- Avoid recomputing values by only computing $Df(\mathbf{x}_k)$ and $D^2f(\mathbf{x}_k)$ once for each k.

The *Rosenbrock function* is a common test function for optimization methods.

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

The minimizer is $\mathbf{x}^* = (1, 1)$ with minimum value $f(1, 1) = 0$. Test your function by minimizing the Rosenbrock function using an initial guess $\mathbf{x}_0 = (-2, 2)$. The function and its derivatives are implemented as `rosen()`, `rosen_der()`, and `rosen_hess()` in `scipy.optimize`. Compare your results to `scipy.optimize.fmin_bfgs()`.

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

>>> f = opt.rosen                    # The Rosenbrock function.
>>> df = opt.rosen_der               # The first derivative.
>>> d2f = opt.rosen_hess             # The second derivative (Hessian).
>>> opt.fmin_bfgs(f=f, x0=[-2,2], fprime=df, maxiter=50)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 35
         Function evaluations: 42
         Gradient evaluations: 42
array([ 1.00000021,  1.00000045])
```

BFGS

Newton's method enjoys quadratic convergence when the initial guess is good enough. However, computing and inverting the Hessian matrix at each step of (12.4) is often prohibitively expensive. The idea behind *quasi-Newton methods* is to numerically approximate the inverse of the Hessian at each step. These methods sacrifice some convergence properties in exchange for becoming less computationally expensive. They also make it possible to optimize functions where D^2f is unknown.

Broyden's method is a high-dimensional generalization of the secant method. Just as the secant method approximates the second derivative of f in (12.2) by using the first derivative at nearby

points, Broyden's method uses the first derivative to update an approximated Hessian matrix.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k^{-1} Df(\mathbf{x}_k)^\mathsf{T}, \qquad A_{k+1} = A_k + \frac{\mathbf{y}_k - A_k\mathbf{s}_k}{\|\mathbf{s}_k\|^2}\mathbf{s}_k^\mathsf{T}, \tag{12.5}$$

where $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = Df(\mathbf{x}_{k+1})^\mathsf{T} - Df(\mathbf{x}_k)^\mathsf{T}$.

Though this method no longer explicitly calculates the Hessian $D^2 f(\mathbf{x}_k)$, it still involves a matrix inversion. The *Sherman-Morrison-Woodbury* formula translates the update rule for A_k in (12.5) into the following update rule for A_k^{-1}.

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

Unfortunately, even if $D^2 f(\mathbf{x}_k)$ is positive definite (which is desirable for minimization), the first-order approximation A_k is not guaranteed to be positive definite, so Broyden's method is unreliable. The *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method remedies this problem by using the following positive definite second-order approximation for the Hessian.

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

The Sherman-Morrison-Woodbury formula can also be applied in this situation to yield a computationally efficient form of BFGS.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - A_k^{-1} Df(\mathbf{x}_k)^\mathsf{T} \tag{12.6}$$

$$A_{k+1}^{-1} = A_k^{-1} + \frac{(\mathbf{s}_k^\mathsf{T}\mathbf{y}_k + \mathbf{y}_k^\mathsf{T}A_k^{-1}\mathbf{y}_k)\mathbf{s}_k\mathbf{s}_k^\mathsf{T}}{(\mathbf{s}_k^\mathsf{T}\mathbf{y}_k)^2} - \frac{A_k^{-1}\mathbf{y}_k\mathbf{s}_k^\mathsf{T} + \mathbf{s}_k\mathbf{y}_k^\mathsf{T}A_k^{-1}}{\mathbf{s}_k^\mathsf{T}\mathbf{y}_k} \tag{12.7}$$

Here $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = Df(\mathbf{x}_{k+1})^\mathsf{T} - Df(\mathbf{x}_k)^\mathsf{T}$ as before.

Problem 2. Write a function that accepts a function $Df : \mathbb{R}^n \to \mathbb{R}^n$, a starting point $\mathbf{x}_0 \in \mathbb{R}^n$, an integer `maxiter`, and a stopping tolerance `tol`. Use BFGS as given in (12.6) and (12.7) to optimize f, with $A_0^{-1} = I$ (the $n \times n$ identity matrix) as the initial approximation to the inverse of the Hessian. Return the final estimate $\mathbf{x}_k$, whether or not the method converged, and the number of iterations computed.

This method is a little tricky and can have issues if $\mathbf{x}_0$ is chosen poorly. Consider the following as you implement your function.

- Use the same stopping criteria as in Problem 1, iterating until either $\|Df(\mathbf{x}_k)\|_\infty <$ `tol` or $k >$ `maxiter`. The usual criteria $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| <$ `tol` is **not** a good choice for BFGS.
- Avoid recomputing values by only calculating each $Df(\mathbf{x}_k)$, $\mathbf{s}_k$, $\mathbf{y}_k$, and $\mathbf{s}_k^\mathsf{T}\mathbf{y}_k$ once.
- Note that $\mathbf{s}_k\mathbf{s}_k^\mathsf{T}$, $\mathbf{y}_k\mathbf{s}_k^\mathsf{T}$, and $\mathbf{s}_k\mathbf{y}_k^\mathsf{T}$ are all *outer products* that result in $n \times n$ matrices. Use `np.outer()` instead of `np.dot()` or the `@` operator for these computations. Carefully identify which parts of (12.7) are scalars and which parts are matrices.
- If $(\mathbf{s}_k^\mathsf{T}\mathbf{y}_k)^2 = 0$, terminate the iteration early to avoid dividing by zero.

Test your function on the Rosenbrock function as in Problem 1.

Note

The formula in (12.7) is not the only way to approximate the inverse Hessian. For example, the *Davidon-Fletcher-Powell* (DFP) method uses the following updating scheme.

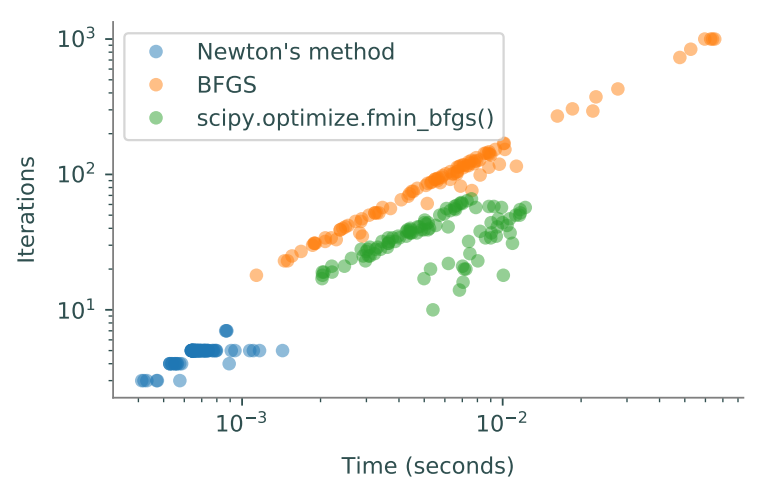
$$A_{k+1}^{-1} = A_k^{-1} + \frac{\mathbf{s}_k\mathbf{s}_k^\mathsf{T}}{\mathbf{s}_k^\mathsf{T}\mathbf{y}_k} - \frac{A_k^{-1}\mathbf{y}_k\mathbf{y}_k^\mathsf{T}A_k^{-1}}{\mathbf{y}_k^\mathsf{T}A_k^{-1}\mathbf{y}_k}$$

This approximation works well for many problems, but BFGS is considered to be the superior method in general.

Problem 3. Write a function that accepts an integer N and performs the following N times.

1. Sample a random initial guess $\mathbf{x}_0$ from the 2-D uniform distribution over $[-3,3]\times[-3,3]$. (Hint: Use `np.random.uniform()` or `np.random.random()`.)
2. Time (separately) your implementation of Newton's method from Problem 1, your BFGS routine from Problem 2, and `scipy.optimize.bfgs_fmin()` for minimizing the Rosenbrock function with an initial guess of $\mathbf{x}_0$.
3. Record the number of iterations from each method. For `scipy.optimize.fmin_bfgs()`, set `disp=False` to suppress printing the convergence message and `retall=True` to get the list of $\mathbf{x}_k$ at each iteration (to count the number of iterations).

Plot the computation times versus the number of iterations with a log-log scale, using different colors for each method. For $N = 100$, your plot should resemble the following figure. Note that Newton's method consistently converges much faster than BFGS. In addition, SciPy's BFGS algorithm will likely converge faster than your BFGS implementation because it employs a line search to choose an intelligent step size at each iteration.



The Gauss-Newton Method

Non-linear Least Squares Problems

Least Squares problems aim to fit a line (or model parameters) to a given set of data points. These problems arise in many scientific fields, including economics, physics, and statistics and represent unconstrained optimization problems that 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 $m \geq n$. This case of least squares problems can be solved with a Newton-like method.

Specifically, with data points $(t_1, y_1), (t_2, y_2), \ldots, (t_m, y_m)$, where $t_i, y_i \in \mathbb{R}$ for $i = 1, \ldots, m$. Let $\phi(\mathbf{x}, \mathbf{t})$ be a possible model for this data set, where $\mathbf{x}$ is a vector of parameters of the model, and $\mathbf{t} \in \mathbb{R}^n$. The error at the i-th data point, called the *residual*, is the value

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

Summing the squares of these errors gives the following non-linear least squares objective function.

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

The first and second derivatives of this function can then be expressed as

$$\begin{aligned} Df(\mathbf{x}) &= J(\mathbf{x})^\mathsf{T} r(\mathbf{x}), \\ D^2 f(\mathbf{x}) &= J(\mathbf{x})^\mathsf{T} J(\mathbf{x}) + \sum_{j=1}^{m} r_j(\mathbf{x}) D^2 r_j(\mathbf{x}). \end{aligned}$$

with $\mathbf{r}(\mathbf{x}) = [r_1(\mathbf{x}), r_2(\mathbf{x}), \ldots, r_m(\mathbf{x})]^\mathsf{T}$ and

$$J(\mathbf{x}) = \begin{bmatrix} \frac{Dr_1(\mathbf{x})}{Dr_2(\mathbf{x})} \\ \hline \vdots \\ \hline Dr_m(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

The second term in the formula for $D^2 f$ involves second derivatives and can be problematic to compute. In practice, the second term in the formula for $D^2 f$ is small, either because the residuals themselves are small, or because they are nearly affine in a neighborhood of the solution. 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

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

The method then proceeds in a manner similar to Newton's method. Thus, at each iteration, we find $\mathbf{x}_{k+1}$ as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (J(\mathbf{x}_k)^\mathsf{T} J(\mathbf{x}_k))^{-1} J(\mathbf{x}_k)^\mathsf{T} \mathbf{r}(\mathbf{x}_k). \tag{12.8}$$

As an example, suppose we have data points generated from the function $y = 3\sin(x/2)$ and slightly perturbed by Gaussian noise. To fit the data to a model $\phi(\mathbf{x}, t_i) = \phi(x_0, x_1, t_i) = x_0 \sin(x_1 t_i)$, we must select values for $\mathbf{x} = [x_0, x_1]^\mathsf{T}$ (since we know how the data was generated, we expect to find that $x_0 \approx 3$ and $x_1 \approx 1/2$). Begin by writing functions for the proposed model, the residual vector, and the Jacobian of the residuals.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

# Generate random data for t = 0, 1, ..., 10.
>>> T = np.arange(10)
>>> y = 3*np.sin(0.5*T)+ 0.5*np.random.randn(10)    # Perturbed data.

# Define the model function and the residual (based on the data).
>>> model = lambda x, t: x[0]*np.sin(x[1]*t)        # phi(x,t)
>>> residual = lambda x: model(x, T) - y            # r(x) = phi(x,t) - y

# Define the Jacobian of the residual function, computed by hand.
>>> jac = lambda x: np.column_stack((np.sin(x[1]*t), x[0]*t*np.cos(x[1]*t)))
```

By inspecting the data, an initial guess for the parameters could be $x_0 = (2.5, 0.6)$. A function implementing Gauss Newton can then be used to find the least squares solution.

```
>>> x0 = np.array([2.5,.6])
>>> x, conv, niters = gauss_newton(jac, residual, x0, maxiter=10, tol=1e-3)

# Plot the fitted model with the observed data and the data-generating curve.
>>> dom = np.linspace(0, 10, 200)
>>> plt.plot(T, y, '*')                          # Observed data.
>>> plt.plot(dom, 3*np.sin(.5*dom), '--')        # Data-generating curve.
>>> plt.plot(dom, model(x, dom))                 # Fitted model.
>>> plt.show()
```

Problem 4. Write a function that accepts a function for the proposed model $\phi(\mathbf{x})$, the model derivative $D\phi(\mathbf{x})$, a function that returns the residual vector $r(\mathbf{x})$, a callable function that returns the Jacobian of the residual $Dr(\mathbf{x}) = J(\mathbf{x})$, a starting point $\mathbf{x}_0$, a max number of iterations `maxiter`, and a stopping tolerance `tol`. This method should implement the Gauss-Newton Method and return a list containing: the minimizing x value, the number of iterations performed, and if the method converged as a boolean.

Test your function by using the Jacobian function, residual function, and starting point given in the example above. Compare your results to `scipy.optimize.leastsq()`.

```
>>> minx = opt.leastsq(func=residual, x0=np.array([2.5,.6]), Dfun=jac)
```

Application of Non-linear Least Squares

Non-linear least squares problems can be used to analyze trends in data or to predict future events and are ubiquitous in many academic fields as well as in industrial applications and machine learning.

Problem 5. The file `population.npy` contains census data from the United States every ten years since 1790 for 16 decades. The first column ($\mathbf{t}$) gives the number of decades since 1790 in the decade $(0, 1, \ldots)$ and the second column ($\mathbf{y}$) gives the population count in millions of people.

By plotting the data, and with a little knowledge about population growth, it is reasonable to hypothesize an *exponential model* for the population:

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

Use the initial guess $(1.5, .4, 2.5)$ for the parameters (x_1, x_2, x_3) and your Gauss Newton function or `scipy.optimize.leastsq()` to fit this model. Plot the resulting curve along with the actual data points.

Unfortunately, the exponential model isn't a very good fit for the data because the population grows exponentially for only the first 8 or so decades.[a] Instead, consider the following *logistic model*.

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

A reasonable initial guess for the parameters (x_1, x_2, x_3) is $(150, .4, -15)$. Write functions for the model and the corresponding residual vector, then fit the model. Plot the data against the fitted curve (in the same plot as before). It should be a much better fit than the exponential curve.

[a] Fitting an exponential model to only the first 8 data points results in a good model for those points (but not for later data).