12 Optimization with Scipy

Lab Objective: *The Optimize package in Scipy provides highly optimized and versatile methods for solving fundamental optimization problems. In this lab we introduce the syntax and variety of `scipy.optimize` as a foundation for unconstrained numerical optimization.*

Numerical optimization is one of the most common modern applications of mathematics. Many mathematical problems can be rewritten in terms of optimization, and unless the problem is both simple and small, usually cannot be solved analytically, and must be approximated by numerical computation.

To assist with this, the `scipy.optimize` package has several functions for minimizing, root finding, and curve fitting, of which we will introduce the most essential.

You can learn about all of the functions at `http://docs.scipy.org/doc/scipy/reference/optimize.html`.

Local Minimization

First we will test out a few of the minimization algorithms on the Rosenbrock function, which is defined as

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

The Rosenbrock function is commonly used when evaluating the performance of an optimization algorithm. Reasons for this include the fact that its minimizer `x = np.array([1., 1.])` is found in curved valley, and so minimizing the function is non-trivial. See Figure 12.1. The Rosenbrock function is included in the optimize package (as `rosen`), as well as its gradient (`rosen_der`) and its hessian (`rosen_hess`).

We will use the `minimize()` function and test some of its algorithms (specified by the keyword argument "method" – see the documentation page for `minimize()`). For each algorithm, you need to pass in a callable function object for the Rosenbrock function, as well as a NumPy array giving the initial guess. For some algorithms, you will additionally need to pass in the jacobian or hessian. You may recognize some of these algorithms, and several of them will be discussed in greater detail later. For this lab, you do not need to understand how they work, just how to use them.

As an example, we'll minimize the Rosenbrock with the Newton-CG method. This method often performs better by including the optional hessian as an argument, which we will do here.

```
>>> import numpy as np
```

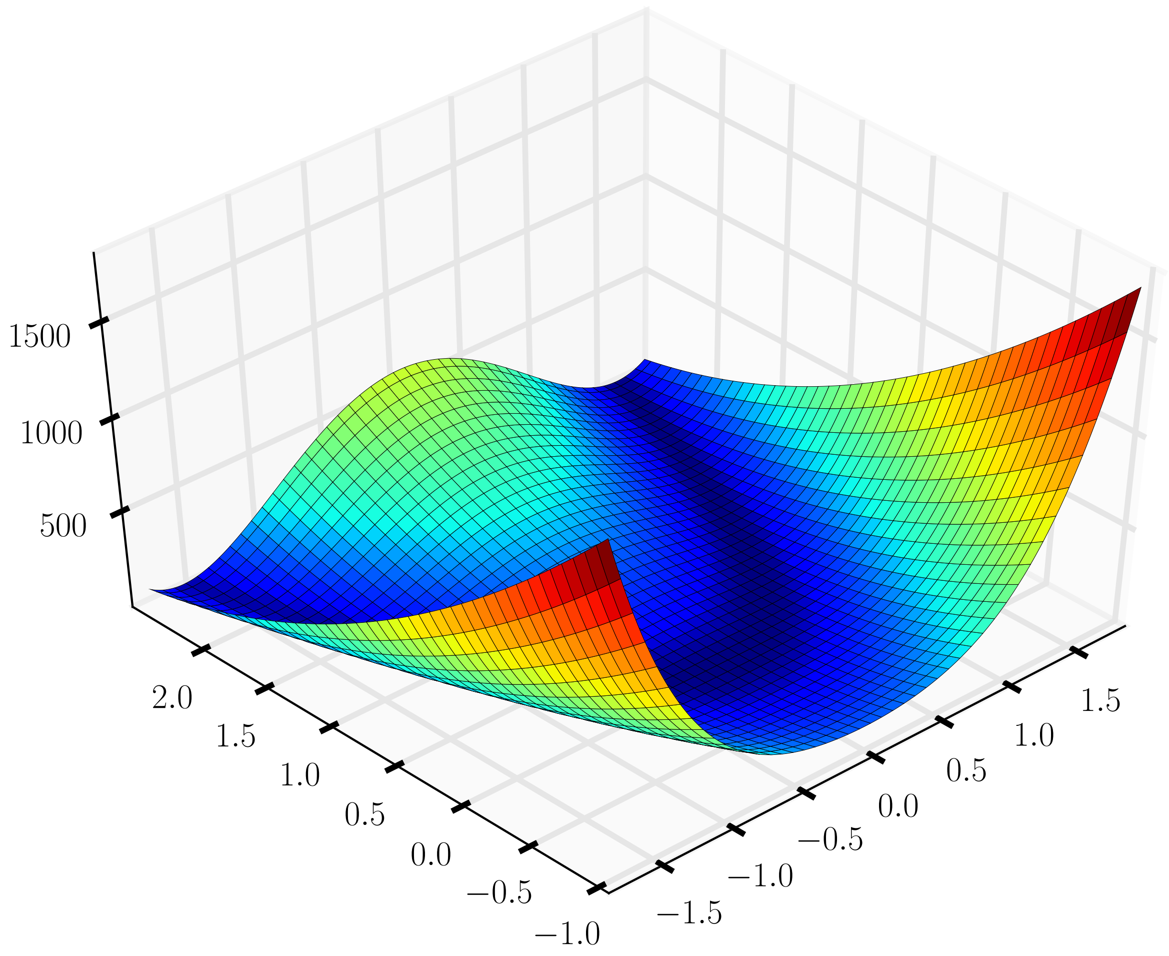

Figure 12.1: $f(x,y) = (1-x)^2 + 100(y-x^2)^2$

```
>>> from scipy import optimize as opt
>>> x0 = np.array([4., -2.5])
>>> opt.minimize(opt.rosen, x0, method='Newton-CG', hess=opt.rosen_hess,
                                                    jac=opt.rosen_der)
     fun: 1.1496545381999877e-15
     jac: array([  1.12295570e-05,  -5.63744647e-06])
 message: 'Optimization terminated successfully.'
    nfev: 45
    nhev: 34
     nit: 34
    njev: 78
  status: 0
 success: True
       x: array([ 0.99999997,  0.99999993])
```

As the online documentation indicates, `opt.minimize()` returns an object of type `opt.optimize.OptimizeResult`.

The printed output gives you information on the performance of the algorithm. The most relevant output for this lab include

`fun: 1.1496545381999877e-15`, the obtained minimum;

`nit: 96`, the number of iterations the algorithm took to complete;

`success: True`, whether the algorithm converged or not;

`x: array([ 0.99999997, 0.99999993])`, the obtained minimizer.

Each of these outputs can be accessed either by indexing `OptimizeResult` object like a dictionary (`result['nit']`), or as attributes of a class (`result.nit`). We recommend access by indexing, as this is consistent with other optimization packages in Python.

The online documenation for `scipy.optimize.minimize()` includes other optional parameters available to users, for example, to set a tolerance of convergence. In some methods, the derivative may be optional, while it may be necessary in others. While we do not cover all possible parameters in this lab, they should be explored as needed for specific applications.

Problem 1. Use the `opt.minimize()` function to find the minimum of the Rosenbrock function. Test Nelder-Mead, CG, and BFGS, starting each with the initial guess `x_0 = np.array([4., -2.5])`. For each method, print whether it converged, and if so, print how many iterations it took.

Note that the hessian argument is not needed for these paticular methods.

Each of these three algorithms will be explored in great detail later in Volume 2: Nelder-Mead is a variation of the Simplex algorithm, CG is a variant of the Conjugate Gradient algorithm, and BFGS is a quasi-Newton method developed by Broyden, Fletcher, Goldfarb, and Shanno.

The `minimize()` function can use various algorithms, each of which is best for certain problems. Which algorithm one uses depends on the specific nature of one's problem.

It is also important to note that in many optimization applications, very little is known about the function to optimize. These functions are often called *blackbox functions*. For example, one may be asked in the airline industry to analyze and optimize certain properties involving a segment of airplane wing. Perhaps expert engineers have designed extremely robust and complicated software to model this wing segment given certain inputs, but this function is so complicated that nobody except the experts dares to try parcing it. Briefly said, you simply don't want to understand it; nobody wants to understand it.

Fortunately, one can still optimize effectively in such situations by wisely selecting a correct algorithm. However, because so little is known about the blackbox function, one must wisely select an appropriate minimization method and follow the specifications of the problem exactly.

Problem 2. Minimize the `blackbox()` function in the `blackbox_function` module. You don't need to know the source code or how it works in order to minimize it. Simply select the appropriate method of `scipy.optimize.minimize()` for this problem, without passing your method a derivative. You may need to test several methods and determine which is most appropriate.

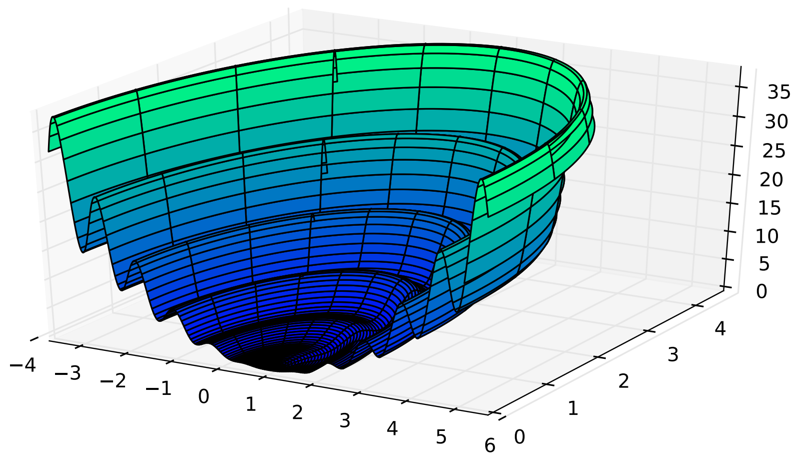

Figure 12.2: $z = r^2(1 + 2\sin^2(4r))$

The function `blackbox()` returns a certain measure of a piecewise-linear curve between two fixed points: the origin, and the point `(40,30)`. This function accepts a one-dimensional `ndarray` of length `m` of y-values, where `m` is the number of points of the piecewise curve excluding endpoints. These points are spaced evenly along the x-axis, so only the y-values of each point are passed into `blackbox()`.

Once you have selected a method, select an initial point with the following code:

```
y_initial = 30*np.random.random_sample(18)
```

Then plot your initial curve and minimizing curve together on the same plot, including endpoints. Note that this will require padding your array of internal y-values with the y-values of the endpoints, so that you plot a total of 20 points for each curve.

Global Minimization via Basin Hopping

In the realm of optimization, convex functions are the most well-behaved, as any local minimum is a global minimum. However, in practice one must frequently deal with non-convex functions, and sometimes we need pick the global minimum out of many local minima.

For example, consider the function

$$z = r^2(1 + \sin^2(4r)),$$

where

$$r = \sqrt{(x+1)^2 + y^2}.$$

Essentially, this is a wavy crater offset from the origin by 1 along the x axis (see Figure 12.2). The presence of many local minima proves to be a difficulty for the minimization algorithms.

For example, if we try using the Nelder-Mead method as previously, with an initial point of `x0 = np.array([-2, -2])`, the algorithm fails to find the global minimum, and instead comes to rest on a local minimum.

```
>>> def multimin(x):
>>>     r = np.sqrt((x[0]+1)**2 + x[1]**2)
>>>     return r**2 *(1+ np.sin(4*r)**2)
>>>
>>> x0 = np.array([-2, -2])
>>> res = opt.minimize(multimin, x0, method='Nelder-Mead')
 final_simplex: (array([[-2.11758025, -2.04313668], [-2.11748198, -2.04319043],
       [-2.11751491, -2.04317242]]), array([ 5.48816866,
                                                      5.48816866,  5.48816866]))
           fun: 5.488168656962328
       message: 'Optimization terminated successfully.'
          nfev: 84
           nit: 44
        status: 0
       success: True
             x: array([-2.11758025, -2.04313668])
>>> print res['x']
[-2.11758025, -2.04313668]
>>> print res['fun']
5.488168656962328
>>> print multimin([-1,0])
0.0
```

However, SciPy does have some tools to help us with these problems. Specifically, we can use the `opt.basinhopping()` function.

The `opt.basinhopping()` function uses the same minimizing algorithms (in fact, you can tell it whatever minimizing algorithm you can pass to `opt.minimize()`). However, once it settles on a minimum, it hops randomly to a new point in the domain (depending on how we set the "hopping" distance) that hopefully lies outside of the valley or basin belonging to the current local minimum. It then searches for the minimum from this new starting point, and if it finds a better minimizer, it repeats the hopping process from this new minimizer. Thus, the `opt.basinhopping()` function has multiple chances to escape a local basin and find the correct global minimum.

Only Scipy Version 0.12+ has `opt.basinhopping`. In earlier versions, such as 0.11, you won't find it.

Problem 3. Explore the documentation for the `opt.basinhopping()` function online or via IPython, and use it to find the global minimum of our `multimin()` function with `x0 = np.array([-2, -2])`. Call it using the same `Nelder-Mead` algorithm with `opt.basinhopping(multimin, x0, stepsize=0.5, minimizer_kwargs={'method':'nelder-mead'})`. Try it first with `stepsize=0.5` and then with `stepsize=0.2`.

Plot the multimin function using the following code:

```
xdomain = np.linspace(-3.5,1.5,70)
```

```
ydomain = np.linspace(-2.5,2.5,60)
X,Y = np.meshgrid(xdomain,ydomain)
Z = multimin((X,Y))
fig = plt.figure()
ax1 = fig.add_subplot(111, projection='3d')
ax1.plot_wireframe(X, Y, Z, linewidth=.5, color='c')
```

Plot the initial point and minima by adapting the following line:

```
ax1.scatter(x_value, y_value, z_value)
```

Why doesn't the alogrithm find the global minimum with `stepsize=0.2`? Print your answer to this question, and return the true global minimum.

Root Finding

The `optimize` package also has functions useful in root-finding. The next example, taken from the online documentation, solves the following nonlinear system of equations using `opt.root`.

$$\begin{bmatrix} x_0 + 1/2(x_0 - x_1)^3 - 1 \\ 1/2(x_1 - x_0)^3 + x_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

```
>>> def func(x):
>>>     return [x[0] + 0.5 * (x[0] - x[1])**3 -1.0,
>>>             0.5 * ( x[1] - x[0])**3 + x[1]]
>>> def jac(x):
>>>     return np.array([[1 + 1.5 * (x[0] - x[1])**2,
>>>                       -1.5 * (x[0] - x[1])**2],
>>>                       [-1.5 * (x[1] - x[0])**2,
>>>                       1 + 1.5 * (x[1] - x[0])**2]])
>>> sol = opt.root(func, [0, 0], jac=jac, method='hybr')
>>> print sol.x
[ 0.8411639  0.1588361]
>>> print func(sol.x)
[-1.1102230246251565e-16, 0.0]
```

Problem 4. Find the roots of the system

$$\begin{bmatrix} -x + y + z \\ 1 + x^3 - y^2 + z^3 \\ -2 - x^2 + y^2 + z^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

Return the values of x, y, z as an array.

As with `opt.minimize()`, `opt.root()` has more than one algorithm for root finding. Here we have used the `hybr` method. There are also several algorithms for scalar root finding. See the online documentation for more.

Curve Fitting

SciPy also has methods for curve fitting wrapped by the `opt.curve_fit()` function. Just pass it data and a function to be fit. The function should take in the independent variable as its first argument and values for the fitting parameters as subsequent arguments. Examine the following example from the online documentation.

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

>>> #the function with which to create the data and later fit it
>>> def func(x,a,b,c):
>>>     return a*np.exp(-b*x) + c

>>> #create perturbed data
>>> x = np.linspace(0,4,50)
>>> y = func(x,2.5,1.3,0.5)
>>> yn = y + 0.2*np.random.normal(size=len(x));

>>> #perform the fit
>>> popt, pcov = opt.curve_fit(func,x,yn)
```

The variable `popt` now contains the fitted parameters and `pcov` gives the covariance of the fit. See Figure 12.3 for a plot of the data and the fitted curve.

One of the most fundamental phenomena in the physical and engineering sciences is turbulent convection, wherein an unstable density gradient induces a fluid to move chaotically (basically, hot air rises). This problem is so important that experiments and numerical simulations have been pushed to their limits in the past several decades to determine the qualitative nature of the fluid's motion under an extreme forcing (think of boiling a pot of water, but instead at temperatures akin to the interior of the sun). The strength of the forcing (amount of the enforced temperature gradient) is measured by the non-dimensional Rayleigh number R. Of paticular interest is to determine how well the chaotic turbulent flow transports the heat from the hot bottom to the cold top, as measured by the Nusselt number ν. One of the primary goals of experiments, simulations, and analysis is to determine how the Nusselt number ν depends on the Rayleigh number R, i.e. if the bottom of the pot of water is heated more strongly, how much faster does the boiling water transport heat to the top?

It is often generically believed that the Nusselt number obeys a power law of the form $\nu = cR^{\beta}$, where $\beta \leq 1/2$. Through some mild assumptions on the temperature, we can construct an eigenvalue problem that we solve numerically for a variety of Rayleigh numbers, thus obtaining an upper bound on the Nusselt number as $\nu \leq cR^{\beta}$. With our physical specifications of the problem, we may predict $\beta < 1/2$.

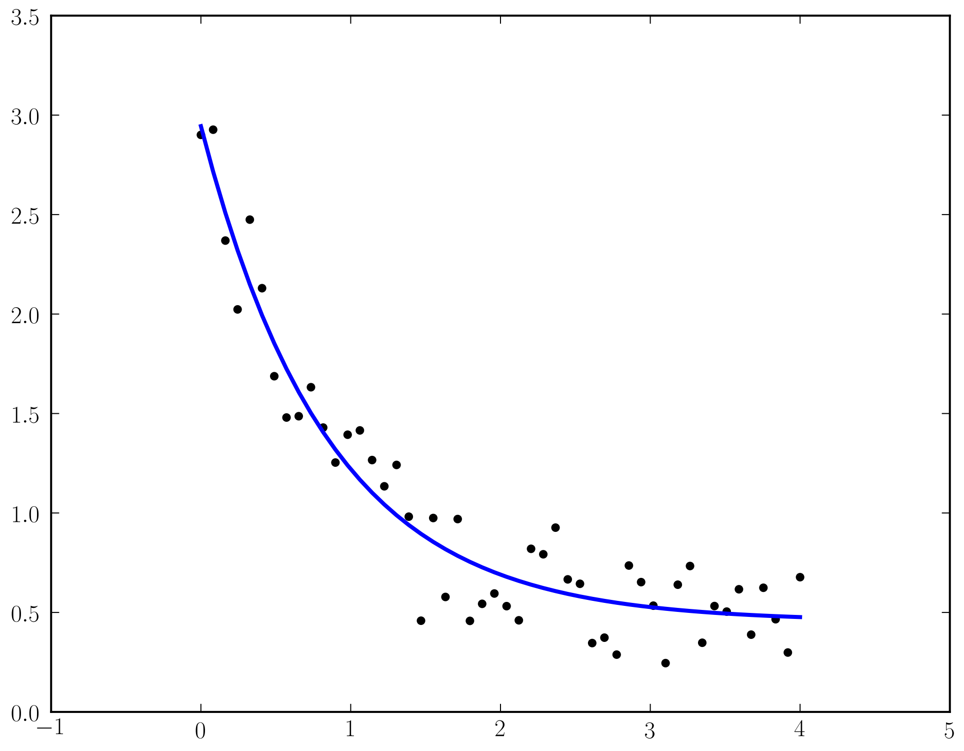

Figure 12.3: Example of perturbed data graphed with the resulting curve using the fitted parameters: $a = 2.72$, $b = 1.31$, and $c = 0.45$.

Problem 5. Use `opt.curve_fit()` to fit a curve to data obtained from numerical simulations of convection. The data are in the file `convection.npy`. The first column is R, the Rayleigh number, and the second column is ν, the Nusselt number. With the convection equation

$$\nu = cR^{\beta},$$

use `opt.curve_fit()` to find a fit to the data using c and β as the fitting parameters. See Figure 12.4 for a plot of the data along with a fitted curve. Though it may be difficult to see in the figure, the first four points skew the data, and do not help us determine the appropriate long-term values of c and β. Thus, do not use the first four points when fitting a curve to the data, but include them in the plot.

Just so that you know that you are getting realistic values, c should be around .1, and β should be less than 1/2. Return your values for c and β in a NumPy array of length 2.

The `scipy.optimize` package has many other useful functions, and is a good first resource when confronting a numerical optimization problem. See the online documentation for further details.

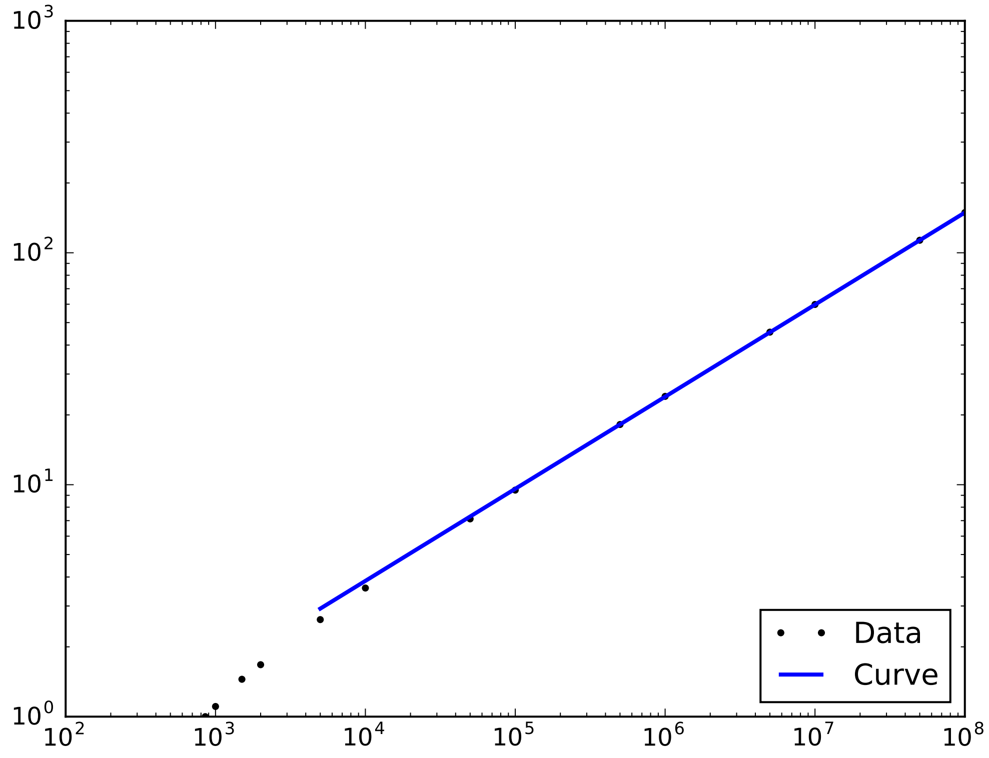

Figure 12.4: The black points are the data from `convection.npy` plotted as a scatter plot. The blue line is a fitted curve.