1. Root-finding problems

An extremely common type of problem is that of *root-finding*. That is, solving the following problem for x:

$$f(x) = 0 \tag{1}$$

There is a related, much harder problem, of solving a multidimensional problem $\vec{x}$, that we will not discuss here; just remember it is much harder!

First, however, note that there are many common cases where the function $f(x)$ is linear. These cases are, obviously, easy. They are even "easy" in multi-dimensions, because they boil down to solving linear systems of equations. What we're interested in here are the harder, nonlinear cases.

A simple example is if I have a routine that tells me the position of the Moon as a function of time, and I want to know when the Moon is going to rise or set next.

2. Bracketing

A critical issue for finding roots is *bracketing*: how to find an interval which contains the root. Any given interval may have more than zero, one, or more roots. In general, if you have a black box function, there may be no root anywhere! As an example, if you are at the North Pole, moon rise or set may be many days away!

A method that is commonly used is to start with some guess as to the bracket, and then just grow it symmetrically by some factor. This may or may not work. In general, the initial bracketing is going to depend a lot on the problem at hand. In my example, I want to know the next moonrise, so I can keep "now" fixed and just extend my other bracket into the future. I also pretty much know that over most of the Earth, the time to the next moon rise or set will be less than 12 hours, and at most it is going to be $\sim$ 13–14 days.

So I give an example here which will work okay for this case. The `scipy.optimize` package has a `bracket` utility which is more general, but also may not be appropriate in all cases. As usual, your mileage may vary!

3. Bisection

By far the most sure-fire route to find the minimum is bisection.

Can you explain to me, or guess, what bisection is?

When should you stop?

How does the accuracy increase at each step of the method?

The `scipy.optimize` utility implements this in the `bisect` routine. It is also fairly easy to implement yourself.

4. Brent's Method

Faster convergence is sometimes desired. As in integration, building some sort of local model for your function can aid in convergence — why blindly take a step if you can guess something about your function?

Brent's Method is a version of this technique. If you are root-finding on $f(x)$, it takes the clever route of fitting a parabola to $x(f)$. You start with the brackets and their midpoint. These three points allow you to fit a parabola in to $x(f)$ and evaluate x at $f = 0$.

This x can be written as:

$$x = b + \frac{P}{Q} \tag{2}$$

where:

$$\begin{aligned} R &= \frac{f(b)}{f(c)} \\ S &= \frac{f(b)}{f(a)} \\ T &= \frac{f(a)}{f(c)} \end{aligned} \tag{3}$$

and:

$$\begin{aligned} P &= S\left[T(R-T)(c-b) - (1-R)(b-a)\right] \\ Q &= (T-1)(R-1)(S-1) \end{aligned} \tag{4}$$

where this particular formulation is designed to avoid instabilities in the quadratic equation. You can verify that the above set of definitions yields the zero for the parabolic fit, but I won't do it here.

If these updates are reasonable, this method will converge somewhere between linearly and quadratically.

At any rate, if we try this, we can easily imagine two types of scenarios: one where the zero point ends up outside the range $a < x < c$; another where the steps are very small (this can happen if the midpoint gets close to one of the end points). In such cases, Brent's method dictates that you take a bisection step instead (between b and whatever bracket point is opposite sign). Thus, your worst case scenario is never much worse than bisection.

`scipy.optimize` implements this as its `brentq` routine.

5. Newton-Raphson

The methods above, and particularly Brent, are what to use if you do not have easy access to the analytic derivative of your function. However, occasionally you do, and the *Newton-Raphson* method takes advantage of this. Also, Newton-Raphson becomes very useful in more than one dimension.

Newton-Raphson simply uses the derivative at one location x to linearly extrapolate to an estimate $x + \delta$ of the zero:

$$\delta = -\frac{f(x)}{f'(x)} \tag{5}$$

We can work out the convergence properties of Newton-Raphson. If we are at $x_i = x + \epsilon$ near the real root x:

$$\begin{aligned} f(x+\epsilon) &= f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2!} f''(x) + \dots \\ f'(x+\epsilon) &= f'(x) + \epsilon f''(x) + \frac{\epsilon^2}{2!} f'''(x) + \dots \end{aligned} \tag{6}$$

Now:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{7}$$

and since x isn't changing:

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)} \tag{8}$$

Plug in the above equations and keep the lowest order in ϵ_i

However, Newton-Raphson can get into some trouble, if it isn't treated carefully, as I show in the notebook.

In general, it is advisable to use bracketing, then Brent's method to some reasonable tolerance, then tune the root using Newton-Raphson if the derivatives are available.

Finally, what if we don't have access to the derivatives? You can forward-difference the derivative, but then it is usually not better to use Newton-Raphson than Brent. You need at least to double your function calls to take a derivative, and that means the convergence power is $\sqrt{2}$, not 2.