Astronomy 8824: Numerical Methods Notes 3
Root-finding and Minimization

Reading: Numerical Recipes, Chapter 9 (Root finding), primarily 9.0, 9.1, and 9.4, with a quick read of 9.2-9.3 and 9.6.

Then Chapter 10 (Minimization), primarily 10.0-10.2 and 10.4, with a glance at 10.3. The later sections of this chapter cover multi-dimensional minimization and are an important reference when you face such a problem, with the Nelder-Mead algorithm described in 10.5 being a slow-ish but robust method. For the specific problem of least-squares minimization, the Levenberg-Marquardt algorithm described in §15.5.2 (3rd edition) is a common choice.

General Remarks

A single linear equation can be solved algebraically. (Duh)

So can a quadratic equation. ($\sqrt{\mathrm{Duh}}$)

A system of non-degenerate linear equations can be solved by the method described in the previous notes, requiring matrix inversion.

A non-linear equation or system of non-linear equations must often be solved numerically, by iteration.

The problem can be expressed in the form

$$f(x) = 0$$

for a single equation or

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

for a system of equations.

The multiple equation problem can be *much* gnarlier than the single equation problem because there is no way to guarantee that you have trapped a root.

Even the 1-d case can have complications, such as multiple roots, or singularities, or pathological cases with roots that are very hard to find.

Whenever possible, you should start by plotting your function so that you have an idea of its global behavior, where you should look for roots, and what could go wrong.

An example

We'll discuss two general algorithms for the 1-d case, but sometimes one has a simple non-linear problem for which there is an easy special-purpose iterative solution.

For example, how do you solve $x + \ln(x) = 7$ if you have a calculator?

In `iPython` try:

```
from math import *
x=7
for iter in range(10):
    print iter,x,x+log(x)
    x+=7-(x+log(x))
```

However, if you try this for 0.7, what happens, and why?

If your iteration method is overcorrecting, so that successive steps are getting bigger rather than smaller and oscillating wildly around the solution, it is sometimes sufficient to just take a fraction of the correction, provided you know that your correction is in the right direction.

For example, try the 0.7 problem with

`x+=0.2*(0.7-(x+log(x)))`.

Bisection

Suppose you know that there is a root of $f(x) = 0$ in the interval (x_1, x_2).

How do you know this? $f(x_1) \times f(x_2) < 0$.

Evaluate $f(x_m)$ where $x_m = (x_1 + x_2)/2$.

Figure out which interval the zero lies in, $x_1 - x_m$ or $x_m - x_2$. Replace x_1 or x_2 with x_m accordingly.

Repeat until you have converged to a desired tolerance.

A reasonable (absolute) tolerance choice may be $\epsilon(x_1 + x_2)/2.$ where ϵ is the machine precision and x_1 and x_2 are your initial brackets, unless you chose them symmetrically about 0.

Bisection converges "linearly," gaining in precision by a constant factor with each iteration (i.e., number of decimal places is proportional to number of iteractions).

This is slow compared to some other methods, but bisection is (a) easy to code, and (b) guaranteed to work even for functions with weird oscillations.

Therefore, if speed is not an issue (e.g., you're not solving an equation thousands or millions of times) it may well be the method of choice.

Newton-Raphson Method

The Newton-Raphson method was not discovered by Newton (1671) or by Raphson (1690) but by Simpson (1740); see Appendix of `arXiv:1203.1034` by Skowron & Gould.

It is powerful for smooth functions where the derivative can also be computed analytically.

At each iteration, Newton-Raphson approximates the function as linear with the known first derivative, then jumps to the root of the linear equation for the next iteration:

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

This method can converge to the root very quickly (quadratically, doubling the number of significant digits with each step).

It can also go disastrously wrong, as is obvious when thinking about a location where $f'(x_i) = 0$.

In general it needs to be combined with some procedure that makes more robust iterations (such as bisection) if Newton-Raphson isn't performing well.

It can be an effective way to "polish" a root once you have gotten close by another method, since a couple of iterations can add a lot of significant digits and many functions are well behaved near their roots.

Secant, False Position, Brent

If you don't have analytic expressions for the derivatives, then it's generally inefficient to evaluate them numerically and use Newton-Raphson.

Instead, one can take a line based on two recent guesses.

This is the secant or false position method (NR §9.2), which differ in which previous point is retained at a new iteration.

This scheme is generally faster than bisection but can go wrong in some cases.

With three previous function values, one can fit a parabola instead of a line to infer the root position.

This inverse quadratic interpolation can be more efficient, but can also go wrong.

Brent's method combines bisection with inverse quadratic interpolation to get the speed of the latter with the robustness of the former.

There are a variety of special techniques for roots of polynomials (which may be complex). See the Skowron & Gould paper referenced above, as well as NR §9.5.

Multiple Dimensions

Multi-dimensional root-finding is generically difficult, for reasons discussed in NR §9.6.

For a smooth function, multi-dimensional Newton-Raphson may work well.

In this case, one gets a system of linear equations instead of a single linear equation, and it can be solved by matrix inversion to get the new multi-dimensional iteration.

In contrast to one dimension, it may be worthwhile doing Newton-Raphson with numerical derivatives.

Minimizing a One-Dimensional Function

As with root-finding, but maybe even more so, you want to know what you can about your function before trying to find a minimum.

The effectiveness of methods can depend a lot on the quality of first guesses.

For floating-point precision ϵ, you can generally only calculate the position of the minimum to within $\sim \sqrt{\epsilon}$ because the function will be parabolic in the neighborhood of the minimum.

The analog to "bullet-proof" bisection is the Golden Section Search.

Start with a triplet of points (a, b, c) that "bracket" a minimum in the sense that $f(a) > f(b)$ and $f(c) > f(b)$ — there's a minimum somewhere between a and b.

At each iteration, choose a new point b' that is a fraction $w = 0.38197$ of the way from b into the larger of the two intervals ab or bc.

This will yield a new triplet in which either b or b' replaces one of the previous outer points.

Why 0.38197? Read the similarity argument in NR §10.1.

One can generally do better than Golden Section by using inverse parabolic interpolation – Brent's method, NR §10.2 – or by making use of derivative information – NR §10.3.

Important: These methods may be guaranteed to find a local minimum, but they are not guaranteed to find the global minimum. Global minimization can be a hard problem even in 1-d. Trying multiple starting points is a useful strategy for at least getting some idea of whether you are finding local rather than global minima.

Multi-Dimensional Minimization

Multi-dimensional minimization is another generically hard problem.

It is particularly hard if there are many local minima.

In general, there is no way to guarantee that there is even a local minimum within some region.

The Downhill Simplex Method, a.k.a. Nelder-Mead algorithm, is a relatively robust and elegant method, based on transforming a simplex to try to in the right direction (NR §10.4).

As with root-finding, other methods using derivatives can be faster, but may be more susceptible to problems with difficult functions.

Markov Chain Monte Carlo techniques, which we'll discuss later, can be a good way of mapping out the minima of a multi-dimensional function.