Notes 4: Root finding algorithms

4.1 Root-finding in one dimension

Given a function, $f(x)$, of a single scalar variable root-finding in one dimension is the task of finding the value or values of x which satisfy the equation

$$f(x) = 0. \tag{4.1}$$

Since most equations cannot be solved analytically, numerical approaches are essential. It is important to remember however that since "most" real numbers do not have exact floating-point representations, we are really interested in finding values of x for which Eq. (4.1) is satisfied approximately in the sense that

$$|f(x)| < \epsilon_{\rm tol} \tag{4.2}$$

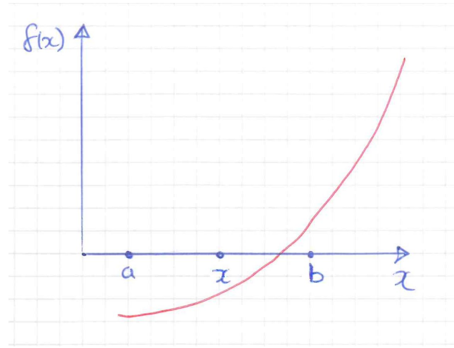
for some pre-specified error tolerance, $\epsilon_{\rm tol}$. All root-finding methods rely on making successive improvements to a reasonable initial guess, x_0, for the position, x_*, of the root. After all, $\mathbb{R}$ is a big place. A root is said to be bracketed by the interval $[a, b]$ if $f(a)f(b) < 0$, that is to say, $f(x)$ has different sign at the two ends of the interval. For continuous functions (and as far as floating-point arithmetic is concerned all functions are continuous), the Intermediate Value Theorem assures the existence of a root somewhere in the interval $[a, b]$. The first step is always to find an interval which brackets the root you are seeking to find. [1] Any reasonable initial guess for the position of the root should be in this interval. Therefore you should always start by plotting your function. This is true for most analytic tasks but especially so for root finding.

There are many algorithms for root finding (and for the related task of finding extremal values fo functions) although we shall only consider a few of them in this module. These algorithms can be usefully divided into two classes: those which use the derivative of the function $f(x)$ and those which do not. Algorithms which use derivatives tend to converge faster but those which do not tend to be more robust. Convergence time is rarely an issue for one-dimensional root finding on modern computers (not necessarily so for higher dimensions) so the increased speed of derivative methods often does not outweigh the sure-footedness of their less sophisticated non-derivative cousins. In any case, it is common to need to analyse functions for which no analytic form for the derivative is known, for example functions which are defined as the outputs of some numerical or combinatorial function.

4.1.1 Derivative-free methods: bracket and bisection method, Brent's method

We start with methods which do not require any knowledge of the derivative. The bracketing-and-bisection algorithm is the most basic root finding method. It involves making successive refinements of the bracketing interval by testing the sign of $f(x)$ at the midpoint of the current interval and then defining a new bracketing interval in the obvious way. At each step, the "best guess" of the position of the root is the midpoint, x:

[1] Note that not every root can be bracketed - a simple counter example is $f(x) = x^2$ which has a root at 0 but no bracketing interval can be chosen. For this reason, the task of finding *all* roots of a nonlinear equation is a-priori a very difficult task.



If we start from a bracketing interval $[a_0, b_0]$, the algorithm consists of the following:

while $b_i - a_i > \epsilon_{\text{tol}}$ **do**
    $x = (a_i + b_i)/2.0$;
    if $f(a_i) * f(x) > 0$ **then**
        $a_{i+1} = x$;
        $b_{i+1} = b$;
    else
        $a_{i+1} = a$;
        $b_{i+1} = x$;
    end
end

What is a reasonable value for ϵ_{tol}? A general rule of thumb is to stop when the width of the bracketing interval has decreased to about $(|a_i| + |b_i|)\,\varepsilon_m/2$ where ε_m is the machine precision. You should think about why that is so.

At each step of the algorithm the width, ϵ_i, of the bracketing interval decreases by a factor of two: $\epsilon_{i+1} = \epsilon_i/2$. Hence, $\epsilon_n = \epsilon_0/2^n$. The number of steps needed to reach accuracy ϵ_{tol} is thus $n = \log_2(\epsilon_0/\epsilon_{\text{tol}})$. While the bracketing-and-bisection method is often said to be "slow" it actually converges exponentially fast! Furthermore it cannot fail.

One could try to improve on simple bracketing-and-bisection with a smarter "best guess" of the position of the root at each step. One way to do this is to fit a quadratic, $P(x)$, through the points $(a_i, f(a_i))$, $(x_i, f(x_i))$ and $(b_i, f(b_i))$ where the current interval is $[a_i, b_i]$ and x_i is the current best guess of the position of the root. This quadratic is used to make the next guess, x_{i+1}, of the position of the root by calculating where it crosses zero. That is, we solve $P(x) = 0$. For a quadratic function this can be done analytically. The quadratic formula involves the calculation of (expensive) square roots and generally produces two roots. For these reasons, it is much better in practice to fit x as a function of y and then evaluate the resulting quadratic at $y = 0$ since this results in a unique value of x and does not require the computation of square roots. This is simple but clever work-around goes by the grandiose name of inverse quadratic interpolation.

The Lagrange interpolating polynomial through the points $(f(a_i), a_i)$, $(f(x_i), x_i)$ and $(f(b_i), b_i)$ is obtained from Eq. (3.1):

$$x = \frac{a_i\,[y - f(x_i)][y - f(b_i)]}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,[y - f(a_i)][y - f(b_i)]}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,[y - f(a_i)][y - f(x_i)]}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.3}$$

Evaluating it at $y = 0$ gives the next best guess for the position of the root:

$$x_{i+1} = \frac{a_i\,f(x_i)\,f(b_i)}{[f(a_i) - f(x_i)][f(a_i) - f(b_i)]} + \frac{x_i\,f(a_i)\,f(b_i)}{[f(x_i) - f(a_i)][f(x_i) - f(b_i)]} + \frac{b_i\,f(a_i)\,f(x_i)}{[f(b_i) - f(a_i)][f(b_i) - f(x_i)]}. \tag{4.4}$$

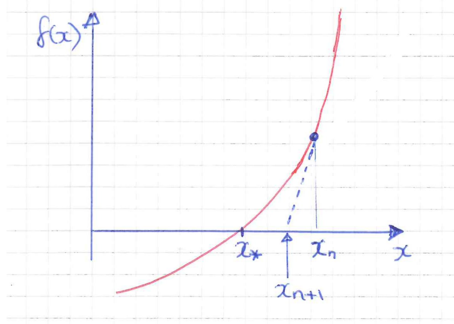
The bracketing interval is then reduced in size as before by testing the sign of $f(x_{i+1})$.

Note that it is possible with this procedure to generate a value for x_{i+1} which lies outside the interval $[a_i, b_i]$. If this happens, one can resort to a simple bisection of the interval as before. Brent's method combines inverse quadratic interpolation with some checks on proposed refinements of the position of the root and some information about previous estimates of the position, using bisection

when necessary, to produce a method which is typically even faster to converge. See [1, chap. 9] for full details. Brent's method is usually the method of choice for generic root-finding problems in one dimension. The idea of inverse quadratic interpolation will come in useful later when we come to the problem of searching for minima.

4.1.2 Derivative methods: Newton-Raphson method

The Newton-Raphson method is the most famous root-finding algorithm. It is a derivative method: it requires the ability to evaluate the derivative $f'(x)$ at any point. The basic idea is the following: at a point, x, which is near to a root, an extrapolation of the tangent line to the curve at x provides an estimate of the position of the root. This is best illustrated geometrically with a picture:



Mathematically, if we are at a point x which is near to a root, x_*, then we wish to find δ such that $f(x+\delta) = 0$. This can be done using Taylor's Theorem, (3.2.1). If x is near to the root then δ is small and we can write

$$0 = f(x+\delta) = f(x) + \delta\, f'(x) + O(\delta^2).$$

Neglecting terms of order δ^2 and above and solving for δ we obtain

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

Our improved estimate for the position of the root is then $x+\delta$. This process can be iterated. If our current estimate of the position of the root is x_i, then the next estimate is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.5}$$

Starting from an initial guess, x_0, the algorithm is the following:

while $|f(x_i)| > \epsilon_{\text{tol}}$ **do**
  $\delta = -\frac{f(x_i)}{f'(x_i)}$;
  $x_{i+1} = x_i + \delta$;
end

If $f'(x)$ is too difficult (or too expensive) to evaluate analytically for use in Eq. (4.5) then the finite difference formulae developed in Sec. 3.2.1 can be used. This is generally to be discouraged since the additional round-off error (recall Fig. 3.5) can badly degrade the accuracy of the algorithm.

The advantage of Newton-Raphson is that it converges very fast. Let us denote the exact position of the root by x_* and the distance from the root by $\epsilon_i = x_i - x_*$. Using Eq. (4.5, we see that

$$\epsilon_{i+1} = \epsilon_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.6}$$

Since ϵ_i is supposed to be small we can use Taylor's Theorem 3.2.1 :

$$\begin{aligned} f(x_* + \epsilon_i) &= f(x_*) + \epsilon_i f'(x_*) + \frac{1}{2}\epsilon_i^2 f''(x_*) + O(\epsilon_i^3) \\ f'(x_* + \epsilon_i) &= f'(x_*) + \epsilon_i f''(x_*) + O(\epsilon_i^2). \end{aligned}$$

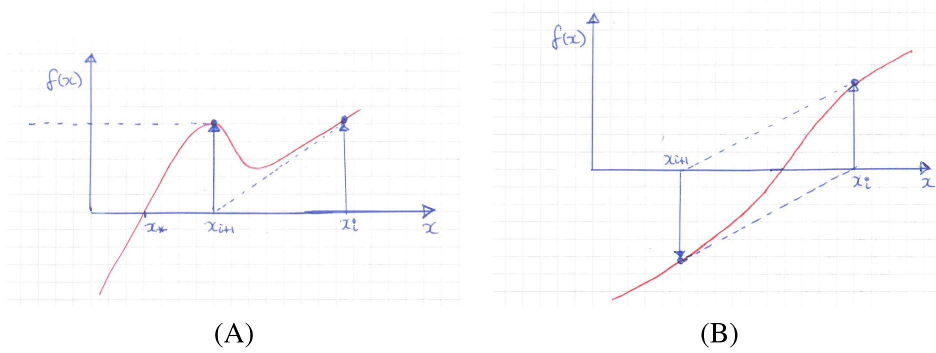


Figure 4.1: Some "unfortunate" configurations where the Newton-Raphson algorithm runs into trouble.

Using the fact that $x_* + \epsilon_i = x_i$ and $f(x_*) = 0$ we can express the x_i-dependent terms in Eq. (4.6) in terms of ϵ_i and the values of the derivatives of $f(x)$ at the root:

$$\begin{aligned} f(x_i) &= \epsilon_i f'(x_*) \left(1 + \frac{\epsilon_i}{2} \frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^3) \\ f'(x_i) &= f'(x_*) \left(1 + \epsilon_i \frac{f''(x_*)}{f'(x_*)}\right) + O(\epsilon_i^2) \end{aligned}$$

Substituting these into Eq. (4.6) and keeping only leading order terms in ϵ_i we see that

$$\epsilon_{i+1} = \epsilon_i^2 \frac{f''(x_*)}{2\, f'(x_*)}.$$

Newton-Raphson iteration therefore converges super-exponentially fast.

The disadvantage of the Newton-Raphson method is that, unlike the bracketing-and-bisection method, it is not guaranteed to converge even if you start near to a bracketed root. Fig. 4.1 shows a couple of unfortunate cases where the Newton-Raphson iteration would fail to converge. In Fig. 4.1(A) an iteration of the method happens to hit on an extremum of $f(x)$. Since $f'(x) = 0$ at an extremum, Eq. (4.5) puts the next guess at infinity! In Fig. 4.1(B), a symmetry between successive iterations sends the method into an infinite loop where the iterations alternate between two values either side of the root.

The potential for complicated dynamics should not come as a surprise to those familiar with the basics of dynamical systems. Eq. (4.5) generalises immediately to complex-valued functions. The result is generally a nonlinear iterated map which can lead to very rich dynamics (periodic cycles, chaos, intermittency etc). For example, Fig. 4.2, shows the basins of attraction in the complex plane of the roots of the polynomial

$$f(z) = z^3 - 1 = 0 \tag{4.7}$$

under the dynamics defined by the Newton-Raphson iteration. One of the great pleasures of nonlinear science is the fact that such beauty and complexity can lurk in the seemingly innocuous task of finding the roots of a simple cubic equation!

4.2 Root-finding in higher dimensions

Root finding in higher dimensions means finding solutions of simultaneous nonlinear equations. For example, in two dimensions, we seek values of x and y which satisfy

$$\begin{aligned} f(x, y) &= 0 \\ g(x, y) &= 0. \end{aligned}$$

There is no analogue to the concept of bracketing a root in higher dimensions. This makes the process of root finding in higher dimensions very difficult. In order to make a guess about the location of a root in two dimensions, there is no real substitute for tracing out the zero level curves of each

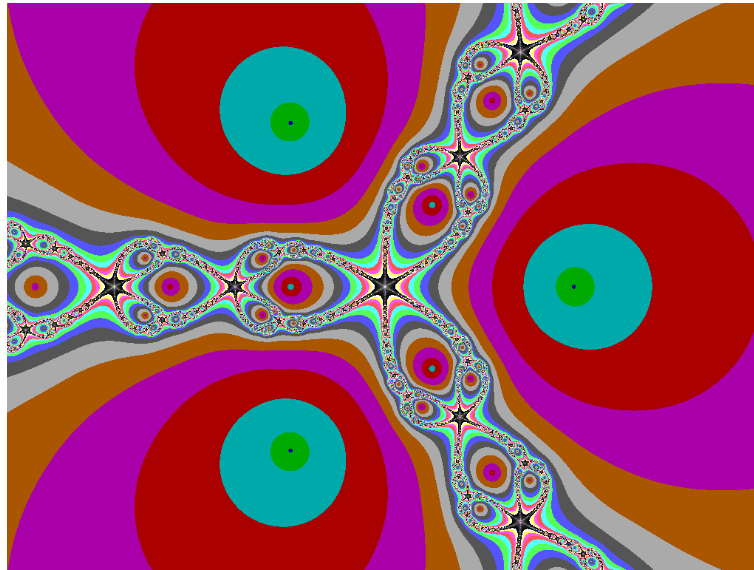


Figure 4.2: (from Wikipedia) Basins of attraction of the roots of Eq. (4.7) in the complex plane. Points are coloured according to the number of iterations of Eq. (4.5) required to reach convergence.

function and seeing if they intersect. In dimensions higher than two, the task rapidly starts to seem like searching for a "needle in a haystack". If, however, one does have some insight as to the approximate location of a root in order to formulate a reasonable initial guess, the Newton-Raphson method does generalise to higher dimensions.

Consider the n-dimensional case. We denote the variables by the vector $\mathbf{x} \in \mathbb{R}^n$. We need n functions of $\mathbf{x}$ to have the possibility of isolated roots. We denote these functions by $F_0(\mathbf{x}) \ldots F_{n-1}(\mathbf{x})$. They can be combined into a single vector valued function, $\mathbf{F}(\mathbf{x})$. We seek solutions of the vector equation

$$\mathbf{F}(\mathbf{x}) = 0. \tag{4.8}$$

If the current estimate of the position of the root is $\mathbf{x}_k$ and we take a step $\boldsymbol{\delta}$ then we can expand the ith component of $F(\mathbf{x}_k + \boldsymbol{\delta})$ using the multivariate generalisation of Taylor's Theorem:

$$F_i(\mathbf{x}_k + \boldsymbol{\delta}) = F_i(\mathbf{x}_k) + \sum_{i=0}^{n-1} J_{ij}(\mathbf{x}_k)\, \delta_j + O(\delta^2) \tag{4.9}$$

where

$$J_{ij}(\mathbf{x}) = \frac{\partial F_i}{\partial x_j}(\mathbf{x})$$

is the (i, j) component of the Jacobian matrix, $\mathbf{J}$, of $\mathbf{F}$ evaluated at $\mathbf{x}$. We wish to choose $\boldsymbol{\delta}$ so that $\mathbf{x}_k + \boldsymbol{\delta}$ is as close as possible to being a root. Setting $F_i(\mathbf{x}_k + \boldsymbol{\delta}) = 0$ in Eq. (4.9) for each i, we conclude that the components of $\boldsymbol{\delta}$ should satisfy the set of linear equations

$$\mathbf{J}(\mathbf{x}_k)\, \boldsymbol{\delta} = \mathbf{F}(\mathbf{x}_k). \tag{4.10}$$

This set of linear equations can be solved using standard numerical linear algebra algorithms:

$$\boldsymbol{\delta} = \mathrm{LinearSolve}(\mathbf{J}(\mathbf{x}_k), \mathbf{F}(\mathbf{x}_k)). \tag{4.11}$$

As for Newton-Raphson iteration in one dimension, the next estimate for the position of the root is $\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\delta}$. We have already seen how the two-dimensional case of Newton-Raphson iteration for complex-valued functions can already lead to very nontrivial dynamics. In higher dimensions, the scope for non-convergence becomes even greater. Nevertheless, if one starts close to a root, the above algorithm usually works.

Bibliography

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing (3rd ed.). Cambridge University Press, New York, NY, USA, 2007.

[2] Nelder-mead method, 2014. `http://en.wikipedia.org/wiki/Nelder-Mead_method#mediaviewer/File:Nelder_Mead2.gif`.

[3] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. Technometrics, 4(4):441–461, 1962.

[4] J. A. Nelder and R. Mead. A simplex method for function minimization. The Computer Journal, 7(4):308–313, 1965.

[5] S. Singer and J. Nelder. Nelder-mead algorithm. Scholarpedia, 4(7):2928, 2009. `http://www.scholarpedia.org/article/Nelder-Mead_algorithm`.

[6] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. The Computer Journal, 7(2):149–154, 1964.

[7] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994. `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`.