Root Finding (and Optimisation)

M.Sc. in Mathematical Modelling & Scientific Computing, Practical Numerical Analysis

Michaelmas Term 2026, Lecture 1

Root Finding

The idea of root finding is simple — we want to find $\mathbf{x}^\star$ such that $\mathbf{f}(\mathbf{x}^\star) = \mathbf{0}$.

Actually finding such roots is not so simple. For example if $f(x)$ is a cubic polynomial

$$f(x) = ax^3 + bx^2 + cx + d,$$

then

$$\begin{aligned} x = & \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) + \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}} \\ & + \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) - \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}} \\ & - \frac{b}{3a}. \end{aligned}$$

Root Finding

If $f(x)$ is a quintic polynomial

$$f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f,$$

then there is a closed form solution for the roots iff its Galois group is solvable (Galois, 1846).

Closed form solutions to the general root-finding problem may:

- not exist;
- be unstable;
- require evaluation of special functions.

In such cases numerical methods help!

Relation to Optimisation

We know that if we want to maximise or minimise a function $g(\mathbf{x})$ then the maximum or minimum occurs at a turning point of g (or on the boundary of the domain) so that

$$f_i := \frac{\partial g}{\partial x_i} = 0.$$

In other words, we have a root finding problem, $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Univariate Root Finding

Bisection Algorithm

In 1D the problem is:

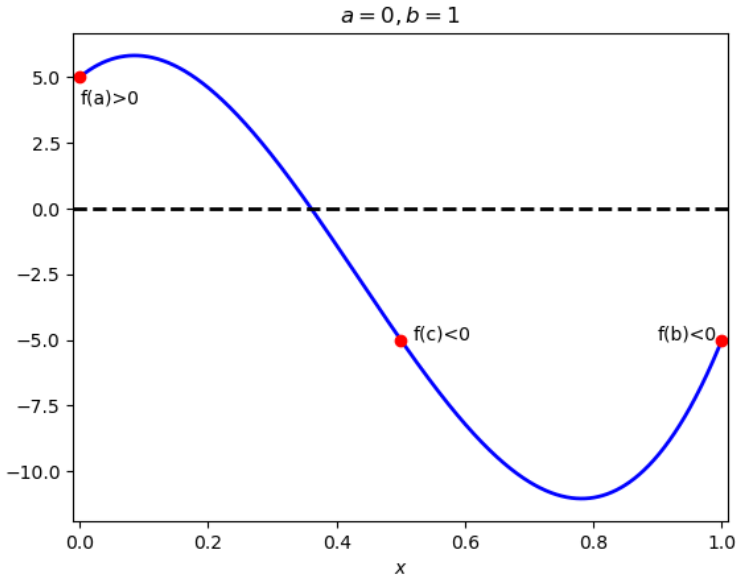
$$\text{Given } f : [a, b] \to \mathbb{R}, \text{ find } c \in [a, b] \text{ such that } f(x^\star) = 0.$$

The bisection algorithm relies on the intermediate value theorem which states:

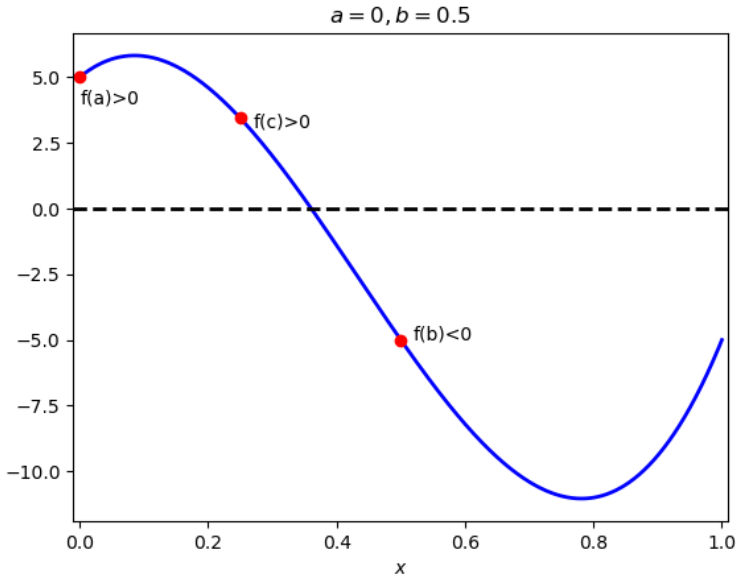
If $f : [a, b] \to \mathbb{R}$ is continuous and $f(a) < u < f(b)$ or $f(b) < u < f(a)$ then $\exists c \in (a, b)$ such that $f(c) = u$.

Thus, to find a root of $f(x)$ we find a and b such that $f(a)$ and $f(b)$ have opposite sign. We set $c = (a + b)/2$, compute $f(c)$ and then repeat on the half of the interval where the sign changes.

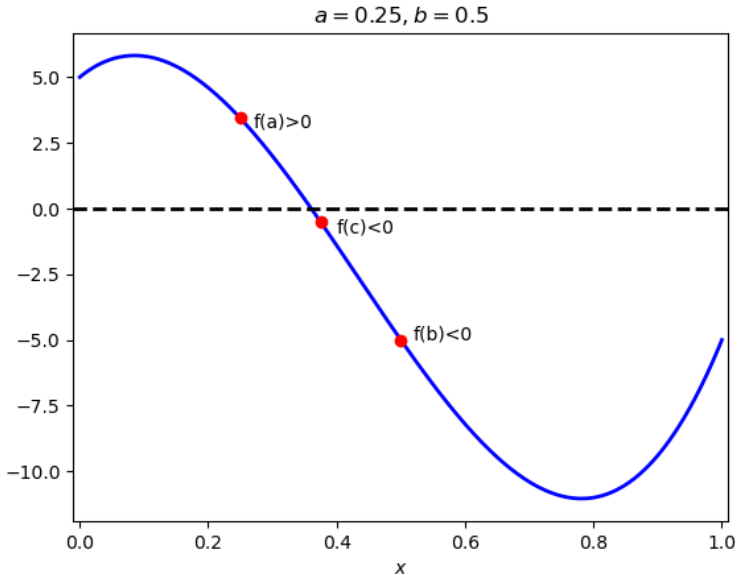
Bisection Algorithm: $f(x) = 100x^3 - 130x^2 + 20x + 5$



Bisection Algorithm: $f(x) = 100x^3 - 130x^2 + 20x + 5$



Bisection Algorithm: $f(x) = 100x^3 - 130x^2 + 20x + 5$



Bisection Algorithm: Termination Criteria

We need to decide when to terminate the algorithm — the normal choices are one of:

- $|b - a| < \text{tol}$ (solution is within a very small interval);
- $|f(c)| < \text{tol}$ (residual is very small).

Bisection Algorithm: Pseudocode

Input: a, b, $f(x)$, tol (end-points of interval containing root, function, error tolerance)
Output: $x^\star$ (approximate root of equation)

1: **while** $|b - a| > \text{tol}$ **do** ▷ or $|f(c)| > \text{tol}$
2: $c \leftarrow (a + b)/2$
3: **if** $f(a)f(c) < 0$ **then** ▷ $f(a)$ and $f(c)$ have opposite sign
4: $b \leftarrow c$
5: **else**
6: $a \leftarrow c$
7: **end if**
8: **end while**
9: $x^\star \leftarrow c$

Bisection Algorithm: Convergence

For convergence, since $c \in [a, b]$, the maximum error in c is $b - a$ (i.e. the width of the interval).

Since each step halves the length of the interval, the error at the nth iteration is

$$|c_n - x^\star| \quad \leq \quad \frac{b-a}{2^n}.$$

To demonstrate convergence numerically note

$$\log|c_n - x^\star| \quad \leq \quad \log(b-a) - n\log(2)$$

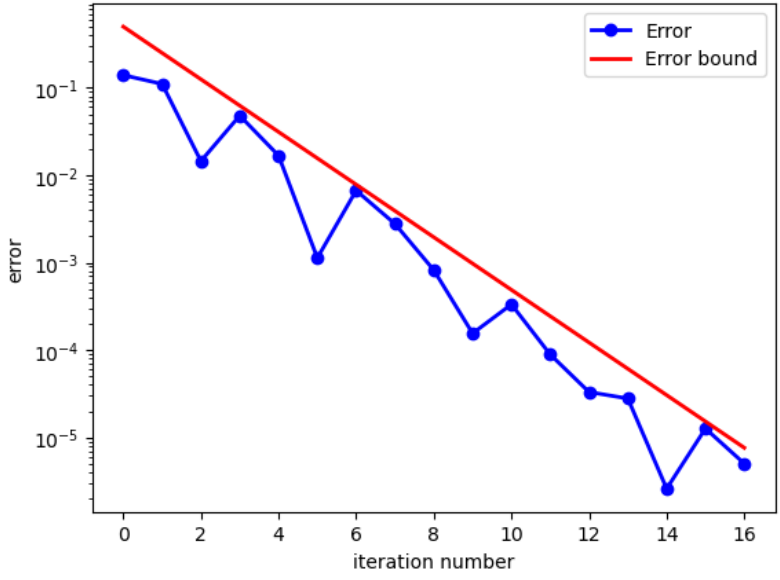
so plotting the log of the error against n gives a straight line with gradient $-\log(2)$.

This also shows that an accuracy of tol ($|c_n - x^\star| \leq$tol) is guaranteed after

$$n \quad \geq \quad \frac{\log(b-a) - \log(\mathrm{tol})}{\log(2)}$$

steps of the bisection algorithm.

Bisection Algorithm: Convergence Example



Regula Falsi

The regula falsi algorithm is similar to the bisection algorithm except that it uses a potentially more intelligent guess for c.

Suppose we approximate the function f on $[a, b]$ by its linear interpolant

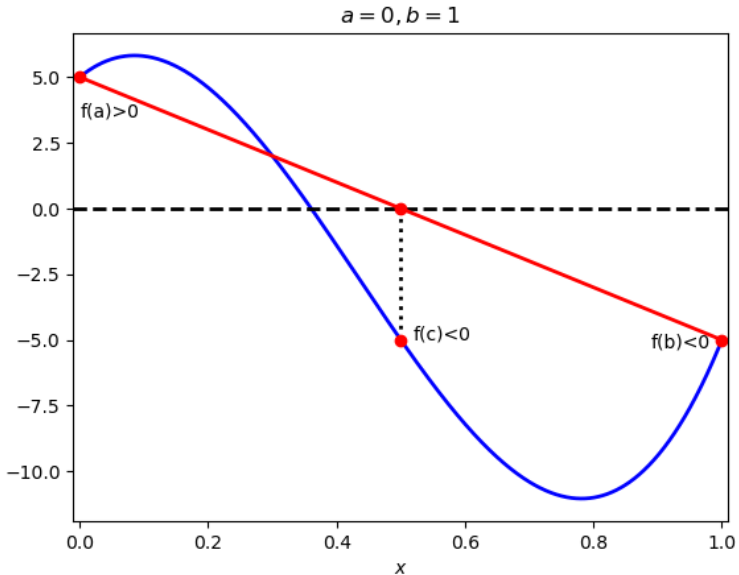
$$p_1(x) = \frac{x-a}{b-a} f(b) + \frac{b-x}{b-a} f(a).$$

Then $p_1(x)$ has a root at

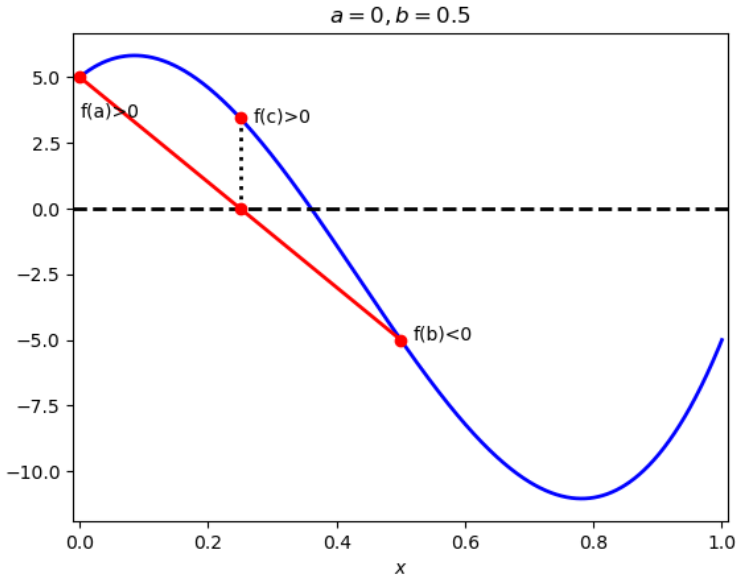
$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}.$$

The regula falsi algorithm uses this as c in the bisection algorithm.

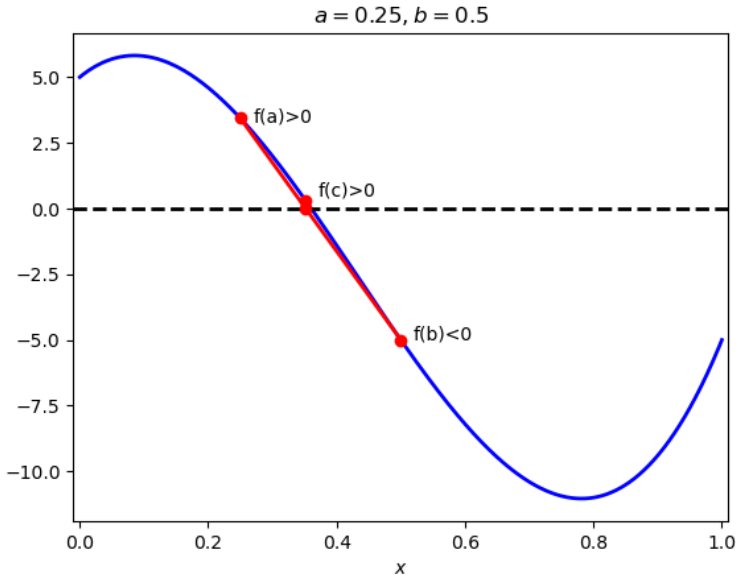
Regula Falsi: $f(x) = 100x^3 - 130x^2 + 20x + 5$



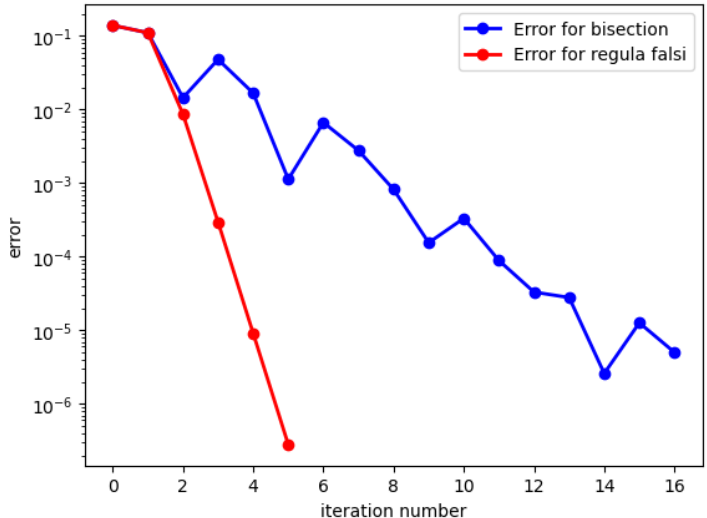
Regula Falsi: $f(x) = 100x^3 - 130x^2 + 20x + 5$



Regula Falsi: $f(x) = 100x^3 - 130x^2 + 20x + 5$



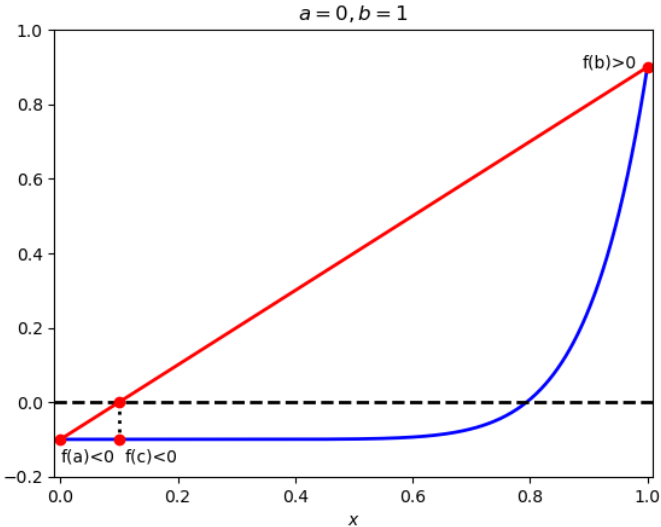
Regula Falsi Convergence



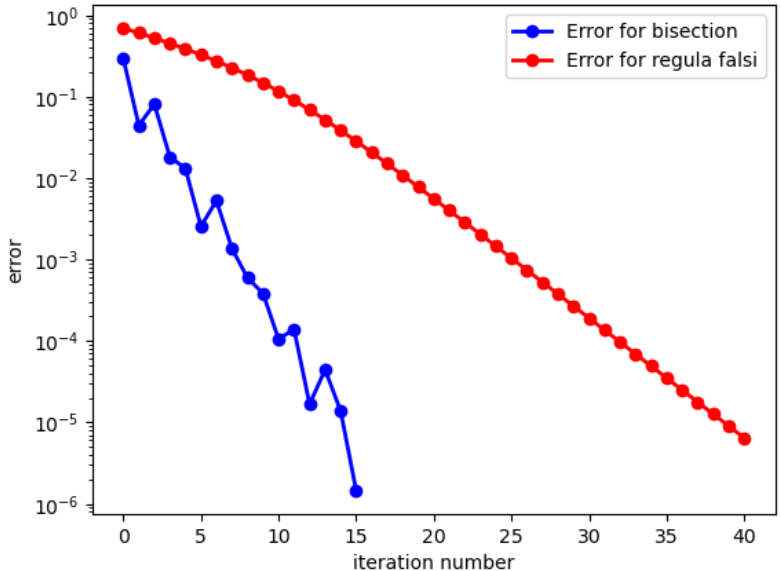
Regula falsi works well when the linear approximation is good.

Regula Falsi: $x^{10} - 1/10$

Convergence can be slow when the curvature of f is large (when the linear approximation is not good).



Regula Falsi: $x^{10} - 1/10$



Regula Falsi: Fix

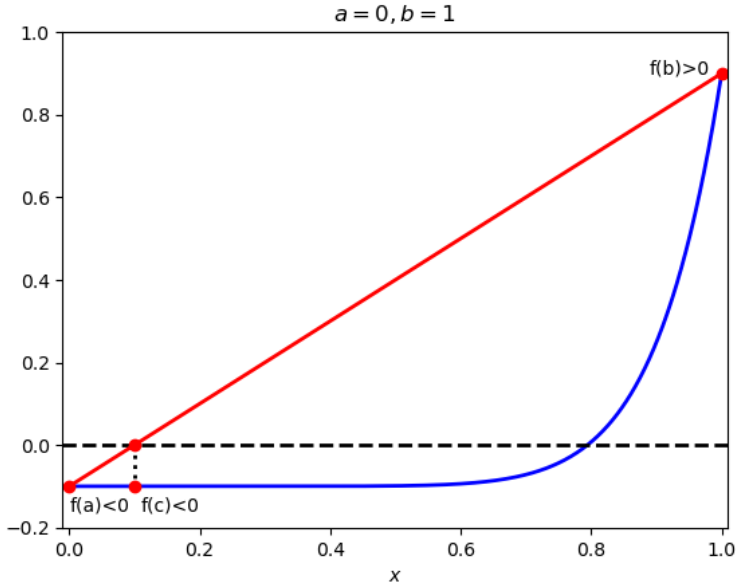
There is a simple fix. If the algorithm goes right twice then halve $f(b)$ so

$$c = \frac{af(b)/2 - bf(a)}{f(b)/2 - f(a)}.$$

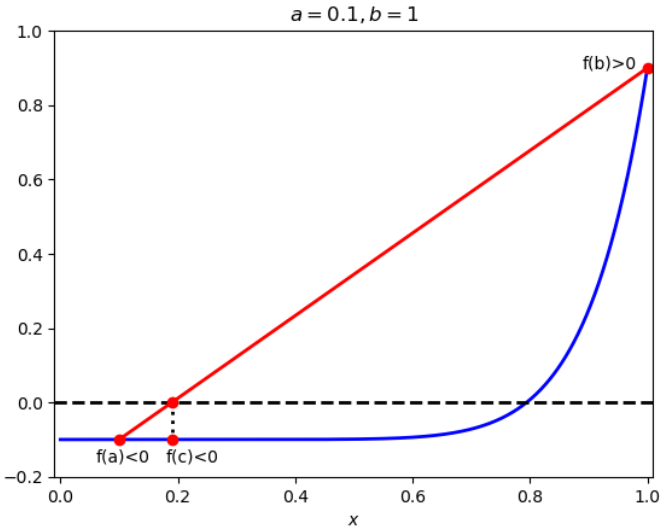
If the algorithm goes left twice then halve $f(a)$. Can show this still gives $c \in (a, b)$.

This method is sometimes called the **Illinois algorithm**.

Illinois Algorithm: $x^{10} - 1/10$

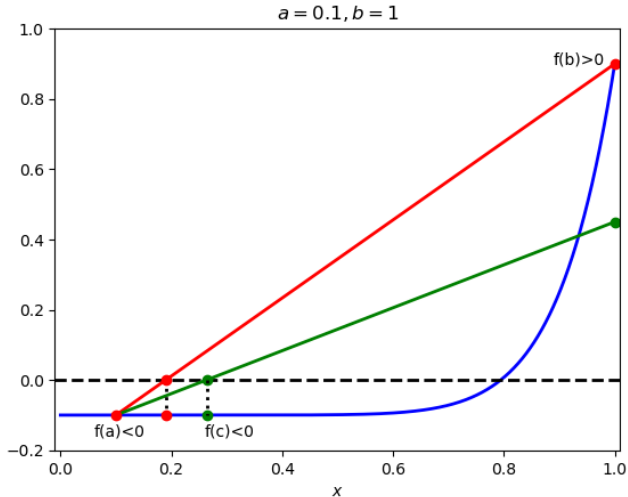


Illinois Algorithm: $x^{10} - 1/10$



After 1 step of regula falsi we want to move right again.

Illinois Algorithm: $x^{10} - 1/10$



Illinois Algorithm (green) has effect of moving further right at the second iteration than standard regula falsi (red).

Newton-Raphson

Suppose we approximate $f(x)$ by $g(x)$ which is the truncated Taylor series of f about the point x_k so

$$g(x) = f(x_k) + (x - x_k)f'(x_k).$$

Then $g(x)$ has a root at c given by

$$c = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Thus the Newton-Raphson method requires an initial guess x_0 and then iterates

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

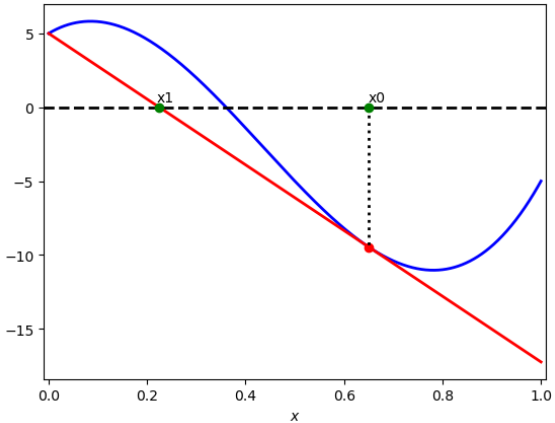
until convergence is achieved.

Newton-Raphson: Geometric Interpretation

We can think of

$$g(x) \quad = \quad f(x_k) + (x - x_k) f'(x_k)$$

as the tangent to $f(x)$ at $x = x_k$. The next iterate is where this tangent meets the x-axis.



Newton-Raphson: Termination Criteria

This time the normal choices for termination are one of:

- $|x_{k+1} - x_k| < \text{tol}$ (solution is only changing by a very small amount);
- $|f(x_k)| < \text{tol}$ (residual is very small).

Newton-Raphson: Pseudocode

Input: x_0, $f(x)$, $f'(x)$, tol (initial guess of root, function and its derivative, error tolerance)

Output: $x^\star$ (approximate root of equation)

1: $k \leftarrow 0$
2: **while** $|f(x_k)| > \mathrm{tol}$ **do**
3: $\quad x_{k+1} \leftarrow x_k - f(x_k)/f'(x_k)$
4: $\quad k \leftarrow k + 1$
5: **end while**
6: $x^\star \leftarrow x_k$

Newton-Raphson: Convergence

If the initial guess is "sufficiently close" to the solution, the algorithm has quadratic convergence.

More precisely, let f be twice continuously differentiable and assume $f(x^\star) = 0$ but $f'(x^\star) \neq 0$. Then Newton's method is locally and quadratically convergent to $x^\star$, i.e. the error at step k, e_k, satisfies

$$\lim_{k \to \infty} \frac{e_{k+1}}{e_k^2} \quad = \quad M,$$

where

$$M \quad = \quad \left| \frac{f''(x^\star)}{2f'(x^\star)} \right|.$$

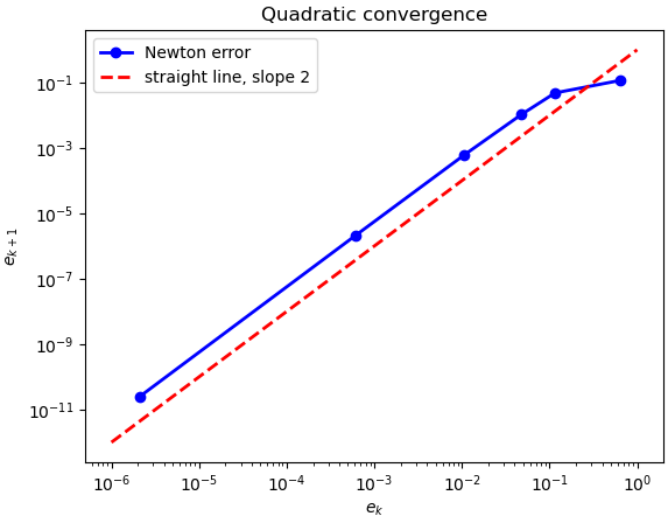
Roughly speaking, we gain 2 digits of accuracy at each iteration once we get sufficiently close to the root.

Newton-Raphson: $f(x) = x^{10} - 1/10$

k	x_k	$e_k = \lvert x_k - x^\star \rvert$	e_k / e_{k-1}^2
0	1.000000	6.39493×10^{-1}	
1	0.910000	1.15672×10^{-1}	0.283
2	0.842368	4.80401×10^{-2}	3.590
3	0.804957	1.06287×10^{-2}	4.605
4	0.794938	6.09806×10^{-4}	5.398
5	0.794330	2.10075×10^{-6}	5.649
6	0.794328	2.50010×10^{-11}	5.665

Newton-Raphson: $f(x) = x^{10} - 1/10$

Since $e_{k+1} \sim Me_k^2$, we have $\log e_{k+1} \sim \log M + 2 \log e_k$, i.e. straight line, gradient 2 on a log-log plot.



Newton-Raphson Modifications

The iterate for the **secant method** is

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

This avoids the need to compute $f'(x_k)$.

For **damped Newton** we use

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

where $\sigma_k \in (0, 1]$.

A third order variation is **Halley's method**:

$$x_{k+1} = x_k - \frac{2f(x_k)f'(x_k)}{2(f'(x_k))^2 - f(x_k)f''(x_k)}.$$

This works if the roots x_* satisfy $f'(x_*) \neq 0$ and is then Newton's method applied to the function $f(x)/\sqrt{|f'(x)|}$.

Polynomial Root Finding

Suppose we want to find the roots of a polynomial

$$p(z) = \sum_{i=0}^{n} a_i z^i.$$

Methods based on bisection will find real roots, one at a time. Methods based on Newton will find roots one at a time. With a complex initial guess, such methods will find complex roots.

In order to find all the roots at once, we can define the *companion matrix* $C \in \mathbb{C}^{n \times n}$ as

$$C = \begin{pmatrix} 0 & & & -a_0/a_n \\ 1 & \ddots & & \vdots \\ & \ddots & 0 & -a_{n-2}/a_n \\ & & 1 & -a_{n-1}/a_n \end{pmatrix}.$$

Polynomial Root Finding

Then the matrix $zI - C$ is given by

$$zI - C = \begin{pmatrix} z & & & a_0/a_n \\ -1 & \ddots & & \vdots \\ & \ddots & z & a_{n-2}/a_n \\ & & -1 & z + a_{n-1}/a_n \end{pmatrix}$$

and we have

$$\det(zI - C) = \frac{p(z)}{a_n}.$$

Thus finding the roots of the polynomial $p(z)$ is equivalent to finding the eigenvalues of the matrix C. The Python command `polyroots` (part of the `numpy.polynomial.polynomial` module) and the Matlab command `roots` use this approach.

Polynomial Root Finding Example

Suppose we want to find the roots of

$$\begin{aligned} p(z) &= (2z-1)(z-1)(2z-3)(z-2)(4z^2+1) \\ &= 6 - 25z + 59z^2 - 120z^3 + 144z^4 - 80z^5 + 16z^6 = \sum_{i=0}^{6} a_i z^i. \end{aligned}$$

```
import numpy as np
from numpy.polynomial.polynomial import polyroots

a = np.array([6, -25, 59, -120, 144, -80, 16])
r1 = polyroots(a)
C = np.diag(np.ones(len(a)-2), k=-1)
C[:, -1] = -a[:-1]/a[-1]
r2 = np.linalg.eigvals(C)
```

r1 and r2 both give a list of roots: $\pm\imath/2$, 0.5, 1, 1.5, 2.

Multivariate Root Finding

Higher Dimensions

The problem is now find to $\mathbf{x}^\star$ such that $\mathbf{f}(\mathbf{x}^\star) = \mathbf{0}$ where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$.

Algorithms based on the bisection algorithm are not extendable to higher dimensions.

Algorithms based on Newton's method are!

Newton's Method

Again we approximate $\mathbf{f}(\mathbf{x})$ by the first terms in its Taylor series:

$$\mathbf{f}(\mathbf{x}+\boldsymbol{\delta}\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + J(\mathbf{x})\boldsymbol{\delta}\mathbf{x},$$

where $J(\mathbf{x})$ is the Jacobian evaluated at the point $\mathbf{x}$,

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}.$$

In order to get the Newton iterate we solve $\mathbf{f}(\mathbf{x}+\boldsymbol{\delta}\mathbf{x}) = \mathbf{0}$, i.e.

$$J(\mathbf{x})\boldsymbol{\delta}\mathbf{x} = -\mathbf{f}(\mathbf{x}),$$

so that

$$J(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = -\mathbf{f}(\mathbf{x}_k).$$

Newton's Method

We can also write the Newton iterate as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (J(\mathbf{x}_k))^{-1}\mathbf{f}(\mathbf{x}_k).$$

Thus a damped version is possible as in 1D:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \sigma_k(J(\mathbf{x}_k))^{-1}\mathbf{f}(\mathbf{x}_k).$$

In both cases we should compute $(J(\mathbf{x}_k))^{-1}\mathbf{f}(\mathbf{x}_k)$ using a linear solver.

Newton's Method and Newton Fractals

One of the problems with using Newton's method is choosing an initial guess. Generally root finding problems have many solutions (think of polynomials) and different initial guesses lead to different solutions with different numbers of iterations. Newton fractals illustrate this behaviour in the complex plane.

Newton Fractals

For example, suppose we consider the function $f(z) = z^3 - 1$. This has three roots at $z = 1$ and $z = (-1 \pm \sqrt{3}\imath)/2$. We can use the 1D Newton's method in the complex variable z to find these roots.

Alternatively, by writing $z = x + \imath y$ and equating real and imaginary parts we get a system

$$\begin{aligned} x^3 - 3xy^2 - 1 &= 0 \\ 3x^2y - y^3 &= 0 \end{aligned}$$

which can be solved using Newton's method.

For each point $z = x + \imath y$ with $(x, y) \in [-2, 2]^2$ we compute a solution using Newton's method and record which root it converges to. A Newton fractal is a plot of this information. Shading using the number of iterations for convergence is also possible.

Newton Fractal Example: $f(z) = z^3 - 1$

