

Numerical Methods

Compiled by Tristan Pang
11/2023

These notes are based on the Oxford Numerical Methods course taught by David Marshall (2023 for the NERC DTP) with additional information from LeVeque [1].

These are rough notes only. A polished version may or may not be completed. Please direct all typos to me. The GitHub repo¹ contains L^AT_EX source, Python scripts for figures, and other useful Python things.

Contents

1	Root finding	2
1.1	Bisection method	2
1.2	Newton's method	3
1.3	Higher dimensions	5
2	Finite difference	5
2.1	Discretising ODEs	5
2.1.1	Forward Euler	6
2.1.2	Backward Euler	6
2.2	Central difference	7
2.3	Leap-frog	8
2.3.1	Robert-Asselin Filter	8
2.4	Mid-point	8
2.5	Trapezoidal	8
2.6	Multi-step	8
2.7	Numerical stability	9
3	Von Neumann analysis	9
4	Numerical linear algebra	9
A	Background theory	9
A.1	Big O notation	9
A.2	Taylor expansions	9

¹<https://github.com/tristanpang/numerical-methods-notes>



Figure 1: Bisection method

References

9

1 Root finding

Consider a sufficiently smooth function $f(x)$. If f is a quadratic polynomial, we may find the zeros of f using the quadratic formula, but for degrees 5 or larger, there exists no general formula for the zeros (Abel–Ruffini theorem). In general, finding an x^* such that $f(x^*) = 0$ cannot be computed exactly. Instead one must employ numerical root finding algorithms. Common methods include the bisection method and Newton’s method.

1.1 Bisection method

To find a zero x^* of f , the bisection method takes two initial guesses a and b such that $a < 0$ and $b \geq 0$. IVT guarantees a zero between the two guesses. Calculate the midpoint

$$c = \frac{a + b}{2}.$$

If $f(c) < 0$, replace a with c ; otherwise replace b with c . Continue iterating until convergence is observed as shown in Figure 1.

The error at the first iteration is

$$\begin{aligned} \varepsilon_1 &= |c - x^*| \\ &= \left| \frac{a - x^*}{2} + \frac{b - x^*}{2} \right| \\ &= \left| \frac{a - x^*}{2} \right| - \left| \frac{b - x^*}{2} \right| \\ &\leq \left| \frac{a - x^*}{2} - \frac{b - x^*}{2} \right| \\ &= \frac{|b - a|}{2}. \end{aligned}$$

Thus, in general

$$\varepsilon_n = |c - x^*| \leq \frac{|b - a|}{2},$$

i.e. the error is at least halved each iteration, and the method converges linearly.

Example 1.1

The positive zero of the polynomial $f(x) = x^2 - 2$ can be approximated using bisection with starting guesses $a = 1$ and $b = 2$. Then $f(1) = -1 < 0$ and $f(2) = 2 > 0$. It follows (by continuity of f) that there is a root in the interval $[1, 2]$. Then $f(c) = f(1.5) = 0.25 > 0$. Thus, we replace $b = 2$ with $c = 1.5$. Continuing yields 1.25, 1.375, 1.4375, ... Eventually, we get an approximation for $\sqrt{2}$.

1.2 Newton's method

A quicker alternative to bisection is Newton's method (also known as Newton-Raphson). Given an initial guess x_0 and a sufficiently nice derivative f' , we may estimate a zero x^* of f .

Consider the Taylor expansion of f around x_n (see Appendix A.1 for the big O notation and Appendix A.2 for Taylor):

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + O((x - x_n)^2).$$

If we suppose that x_n is close to the root x^* , the zero of the linear approximation x_{n+1} is a good approximation for x^*

$$f(x_{n+1}) \approx 0 = f(x_n) + (x_{n+1} - x_n)f'(x_n).$$

Rearranging, we arrive at the iterative formula for Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (1)$$

This process is illustrated in Figure 2.

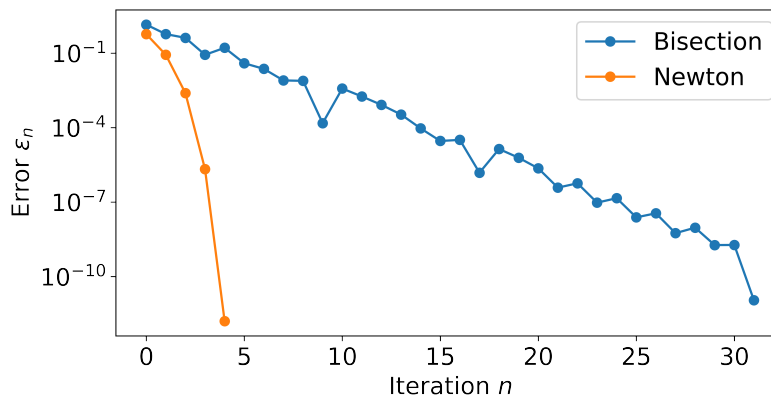
The signed error at iteration n is $\varepsilon_n = x_n - x^*$. By considering the quadratic term in the Taylor expansion around x_n , we get

$$\begin{aligned} f(x^*) &= f(x_n) + f'(x_n)(x^* - x_n) + \frac{f''(x_n)}{2}(x^* - x_n)^2 + O((x^* - x_n)^3) \\ \implies 0 &= f(x_n) + f'(x_n)(x^* - x_n) + \frac{f''(x_n)}{2}\varepsilon_n^2 + O(\varepsilon_n^3) \\ \implies -\frac{f(x_n)}{f'(x_n)} &= (x^* - x_n) + \frac{f''(x_n)}{2}\varepsilon_n^2 + O(\varepsilon_n^3) \\ \implies x_{n+1} - x_n &= (x^* - x_n) + \frac{f''(x_n)}{2}\varepsilon_n^2 + O(\varepsilon_n^3) \\ \implies \varepsilon_{n+1} &= \frac{f''(x_n)}{2}\varepsilon_n^2 + O(\varepsilon_n^3) \end{aligned}$$

Thus, as $n \rightarrow \infty$, $x_n \rightarrow x^*$ for a root x^* of f . In particular, we have quadratic convergence.



Figure 2: Newton's method

Figure 3: Bisection method vs Newton's method for approximating $\sqrt{2}$ **Example 1.2**

The positive zero of the polynomial $f(x) = x^2 - 2$ can be approximated using Taylor's method with the starting guess $x_0 = 2$. Differentiating, $f'(x) = 2x$. Then we get

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{2^2 - 2}{4} = 1.5,$$

$$x_2 = 1.5 - \frac{1.5^2 - 2}{3} = 1.41666667,$$

$$x_3 = 1.41421569,$$

$$x_4 = 1.41421356 \approx \sqrt{2}.$$

This converges to $\sqrt{2}$ much faster than the bisection method as seen in Figure 3.

Exercise 1.3

Observe (in Python or otherwise) that approximating $\sqrt{2}$ with a bisection guess of (1, 200) and Newton guess of 200 yields similar log-linear error behaviour for small iteration step n . Show that this is true by looking at Formula 1.

Warning: if $f'(x_n) = 0$, Newton's method will not work (division by zero!) – pick a new x_0 .

If the derivative is not well behaved (either not defined or close to zero at many points), then Newton's method may not be appropriate.

1.3 Higher dimensions

Consider the system of m equations in n variables $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ given by

$$\begin{cases} f_1(x_1, x_2, \dots, x_n) = 0, \\ f_2(x_1, x_2, \dots, x_n) = 0, \\ \vdots \\ f_m(x_1, x_2, \dots, x_n) = 0. \end{cases}$$

The iterative step of Newton's method becomes

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J(\mathbf{x}_n)^{-1} \mathbf{f}(\mathbf{x}_n), \quad (2)$$

where J is the Jacobian matrix of \mathbf{f} (an analogue to the derivative) given by $J_{ij} = \frac{\partial f_i}{\partial x_j}$. This requires either matrix inversion (which is usually hard!) or solving a linear system (see Section 4).

Example 1.4

Consider the steady state of the predator prey model:

$$\begin{cases} f_1(x, y) = Ax - Bxy = 0, \\ f_2(x, y) = Dxy - Cy = 0. \end{cases}$$

The Jacobian is

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} = \begin{pmatrix} A - By & -Bx \\ Dy & Dx - C \end{pmatrix}.$$

Let $\mathbf{x}_0 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$. Then

$$\mathbf{x}_1 = \mathbf{x}_0 - J(\mathbf{x}_0)^{-1} \mathbf{f}(\mathbf{x}_0) = \begin{pmatrix} 2 \\ 1 \end{pmatrix} - \begin{pmatrix} A - B & -2B \\ D & 2D - C \end{pmatrix}^{-1} \begin{pmatrix} 2A - 2B \\ 2D - C \end{pmatrix}.$$

Note 1.5 (Useful commands)

- Python SciPy's `optimize.fsolve` finds the roots of a function.
- Python NumPy's `linalg.solve` solves a linear system.

2 Finite difference

2.1 Discretising ODEs

Consider the first order differential equation

$$\frac{dy}{dt} = \mathbf{f}(\mathbf{y}, t).$$

The Taylor expansion of \mathbf{y} about time t is

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \frac{d\mathbf{y}}{dt}\Delta t + O((\Delta t)^2).$$

Rearranging and dividing through by Δt yields the first order accurate finite difference approximation

$$\frac{d\mathbf{y}}{dt} = \frac{\mathbf{y}(t + \Delta t) - \mathbf{y}(t)}{\Delta t} + O(\Delta t). \quad (3)$$

2.1.1 Forward Euler

Let $t_{n+1} = t_n + \Delta t = t_0 + n\Delta t$. Given $\mathbf{y}(t_n)$, we can step forward one time step by substituting Equation 3 into the differential equation

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + \mathbf{f}(\mathbf{y}(t_n), t_n)\Delta t + O((\Delta t)^2).$$

The local truncation error is then

$$\tau_{n+1} = (\mathbf{y}(t_{n+1}) - \mathbf{y}(t_n)) - \mathbf{f}(\mathbf{y}(t_n), t_n)\Delta t = O((\Delta t)^2).$$

(Note that LeVeque calls $\frac{\tau_{n+1}}{\Delta t}$ the local truncation error.) Since the total number of steps is proportional to $\frac{1}{\Delta t}$, the global error is $\frac{\tau_{n+1}}{\Delta t} = O(\Delta t)$. This tells us that the forward Euler method is first order accurate. Since the next time step can be calculated directly from the information from the current time step, forward Euler is an explicit method.

2.1.2 Backward Euler

If we take a backward difference instead of a forward difference, we get

$$\mathbf{y}(t_n) = \mathbf{y}(t_{n+1}) - \mathbf{f}(\mathbf{y}(t_{n+1}), t_{n+1})\Delta t + O((\Delta t)^2).$$

Since we cannot determine $\mathbf{y}(t_n)$ directly, backward Euler is implicit, and we must use a root finding algorithm.

Example 2.1

Consider the IVP $\frac{dy}{dt} = -ky = f(y, t)$ with initial value $y(t_0) = 1$. This can be integrated exactly and has an analytical solution of $y(t) = \exp(-k(t - t_0))$. Using forward Euler,

$$y(t_1) = y(t_0) + f(y(t_0), t_0)\Delta t = 1 - k\Delta t.$$

Similarly, for backward Euler,

$$y(t_1) = y(t_0) + f(y(t_1), t_1)\Delta t = 1 - ky(t_1)\Delta t \implies y(t_1) = \frac{1}{1 + k\Delta t}.$$

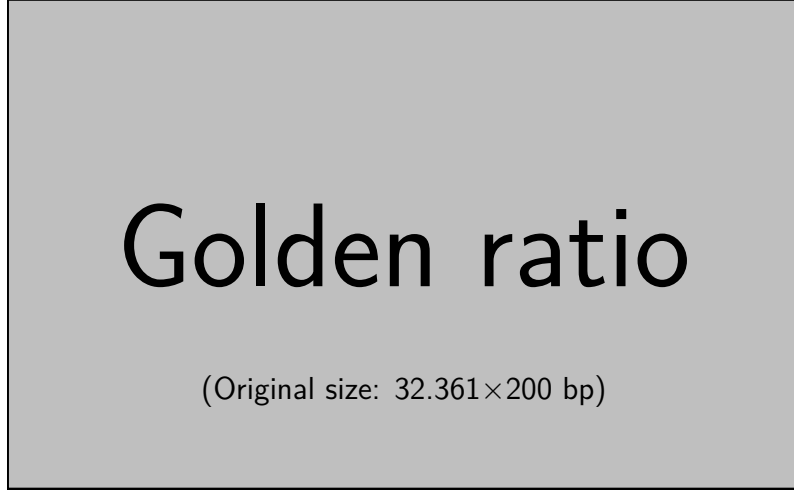


Figure 4: Euler

2.2 Central difference

One way to reduce the truncation error from the first order forward and backward Euler schemes is to use more points in time, say

$$y' \approx ay_{n+1} + by + cy_{n-1},$$

where $y = y(t_n)$, $y_{n+1} = y(t_{n+1})$, $y_{n-1} = y(t_{n-1})$ and $y' = \frac{dy}{dt}|_{t_n}$. To find the best choices of a, b, c , we first Taylor expand

$$\begin{aligned} y_{n+1} &= y + y' \Delta t + \frac{y''}{2} (\Delta t)^2 + \frac{y'''}{6} (\Delta t)^3 + O((\Delta t)^4), \\ y_{n-1} &= y - y' \Delta t + \frac{y''}{2} (\Delta t)^2 - \frac{y'''}{6} (\Delta t)^3 + O((\Delta t)^4). \end{aligned}$$

It follows that

$$ay_{n+1} + by + cy_{n-1} = (a+b+c)y + (a-c)y' \Delta t + (a+c) \frac{y''}{2} (\Delta t)^2 + (a-c) \frac{y'''}{6} (\Delta t)^3 + O((\Delta t)^4).$$

We want $a + b + c = a + b = 0$, i.e. $a = -c$ and $b = 0$. Making the decision $(a - c)\Delta t = 1$, we get $a = \frac{1}{2\Delta t}$ and $c = -\frac{1}{2\Delta t}$. The centred difference of y' is thus

$$\frac{dy}{dt} \Big|_{t_n} = \frac{y_{n+1} - y_{n-1}}{2\Delta t} + O((\Delta t)^2)$$

which is second order accurate.

If instead we took $a + b + c = a - c = 0$ and $(a + c) \frac{(\Delta t)^2}{2} = 1$, then $a = c = \frac{1}{(\Delta t)^2}$ and $b = \frac{-2}{(\Delta t)^2}$. The centred difference for the second derivative is

$$\frac{d^2 y}{dt^2} \Big|_{t_n} = \frac{y_{n+1} - 2y_n + y_{n-1}}{2} + O((\Delta t)^2).$$



Figure 5: Leap-frog

2.3 Leap-frog

$$\mathbf{y}_{n+1} = \mathbf{y}_{n-1} + 2\mathbf{f}(\mathbf{y}_n, t_n)\Delta t + O((\Delta t)^3).$$

Needs forward Euler for the first step.

Needs time filtering to stop the even/odd time iterations diverging.

2.3.1 Robert-Asselin Filter

2.4 Mid-point

Explicit mid-point

Implicit mid-point

2.5 Trapezoidal

(Also called Crank-Nicolson.)

Example 2.2

2.6 Multi-step

AB1 = Forward Euler

2.7 Numerical stability

3 Von Neumann analysis

4 Numerical linear algebra

A Background theory

A.1 Big O notation

A.2 Taylor expansions

Theorem A.1 (Taylor)

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \cdots$$

References

- [1] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007.