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 repo[1] contains LaTeX source, Python scripts for figures, and other useful Python things.

Contents

[1] https://github.com/tristanpang/numerical-methods-notes

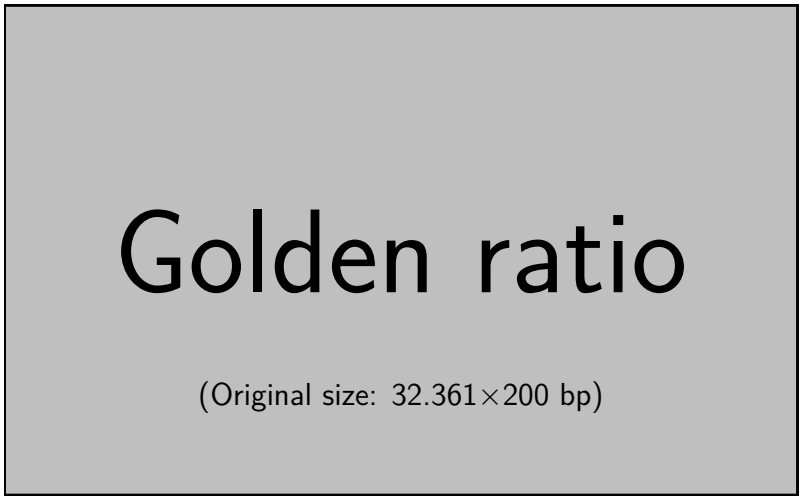

Figure 1: Bisection method

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

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

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 $1.25, 1.375, 1.4375, \ldots$. 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

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

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

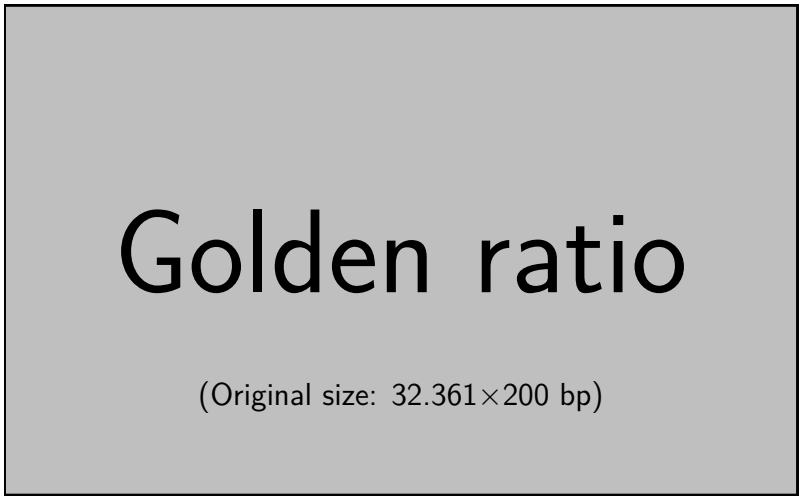

Figure 2: Newton's method

Taylor expansion around x_n, we get

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

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

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

$$\begin{aligned}
x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{2^2 - 2}{4} = 1.5, \\
x_2 &= 1.5 - \frac{1.5^2 - 2}{3} = 1.41666667, \\
x_3 &= 1.41421569, \\
x_4 &= 1.41421356 \approx \sqrt{2}.
\end{aligned}$$

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.

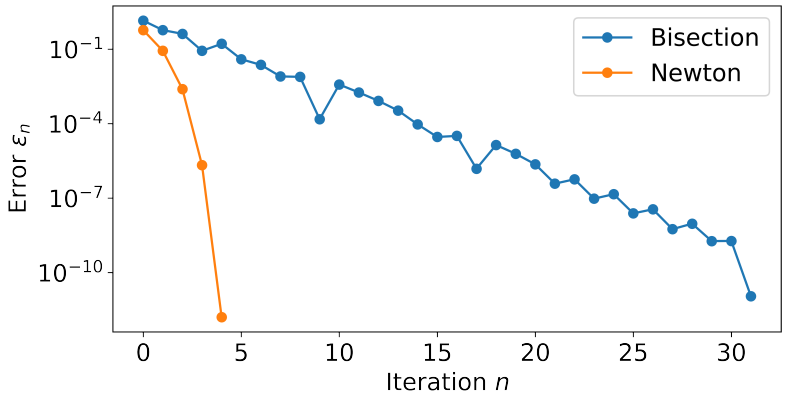

Figure 3: Bisection method vs Newton's method for approximating $\sqrt{2}$

1.3 Higher dimensions

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

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

The iterative step of Newton's method becomes

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

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 5).

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

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

Note 1.5 (Useful commands)

- Python SciPy's `otimize.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

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

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

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

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

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

Letting $\mathbf{y}_{n+1} \approx \mathbf{y}(t_{n+1})$, a forward Euler step is

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

The local truncation error is then

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

(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

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

Letting $\mathbf{y}_{n+1} \approx \mathbf{y}(t_{n+1})$, a backward Euler step is

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

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

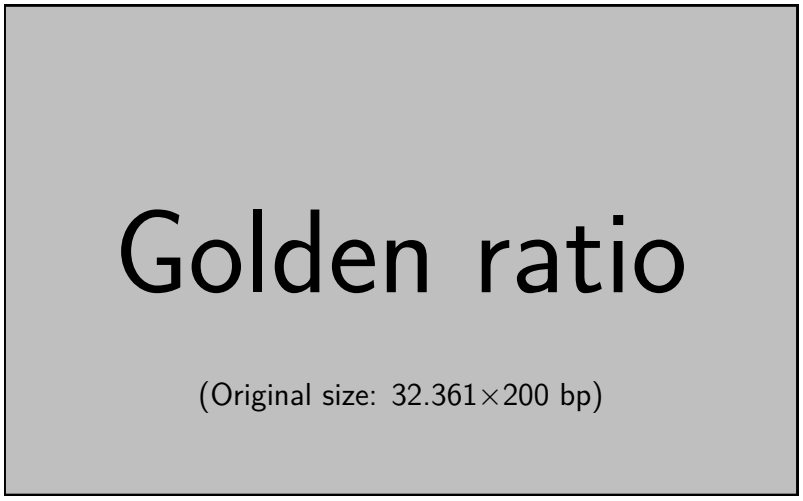

Figure 4: Euler

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,

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

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 $y' = \frac{dy}{dt}\big|_{t_n}$. To find the best choices of a, b, c, we first Taylor expand

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

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

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

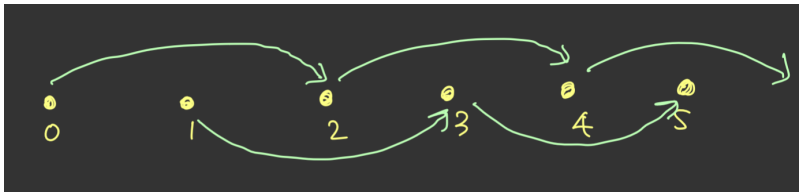

Figure 5: Leap-frog

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

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

2.3 Leap-frog

Leap-frog is a second order explicit method given by

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

We need to use forward Euler for the first step. A stencil is shown in figure 5. Notice that the even and odd iterations are separate, so we need time filtering to stop the even/odd time iterations diverging.

2.3.1 Robert-Asselin Filter

After solving for $\mathbf{y}_{n+1}$, we apply a time filter to $\mathbf{y}_n$ to "smooth" out the scheme. Define

$$\mathbf{y}_n^{\text{new}} = \mathbf{y}_n + \frac{\nu}{2}(\mathbf{y}_{n-1} - 2\mathbf{y}_n + \mathbf{y}_{n+1}),$$

where $0.01 \leq \nu \leq 0.2$ is a smoothing parameter. This adds a small fraction of the two outer points to $\mathbf{y}_n$. We then set $\mathbf{y}_{n+1}$ to be $\mathbf{y}_n^{\text{new}}$ and $\mathbf{y}_{n+1}$ to be $\mathbf{y}_n$ before taking the next time step.

2.4 Other methods

We introduce a few more second order schemes.

2.4.1 Mid-point

Explicit mid-point takes a half forward Euler step before performing the full step

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{f}\left(\mathbf{y}_n + f(\mathbf{y}_n, t_n)\frac{\Delta t}{2}, t_{n+\frac{1}{2}}\right)\Delta t.$$

Implicit mid-point is

$$y_{n+1} = y_n + f\left(\frac{y_n + y_{n+1}}{2}, t_{n+\frac{1}{2}}\right)\Delta t.$$

2.4.2 Trapezoidal

The trapezoidal method (also called Crank-Nicolson) is a combination of forward and backward Euler

$$y_{n+1} = y_n + \left(\frac{f(y_n, t_n) + f(y_{n+1}, t_{n+1})}{2} \right) \Delta t.$$

Example 2.2
Consider the ODE $\frac{dy}{dt} = -kt$ with $y(t_0) = 1$. The explicit mid-point gives

$$y_1 = -k\Delta t + \frac{k}{2}(\Delta t)^2.$$

Since the ODE is linear, the trapezoidal method is equivalent to implicit mid-point

$$y_1 = \frac{1 - \frac{k}{2}\Delta t}{1 + \frac{k}{2}\Delta t}.$$

2.5 Multi-step

The typical family of multi-step methods are the Adams-Bashforth methods. These are explicit. The key idea is to fit a polynomial to $f(t)$ and integrate over the t-step.

2.5.1 AB2

For convenience, set $t = 0$ at y_n. Then

$$y_{n+1} = y_n + \int_0^{\Delta t} f(y_n, t) \, \mathrm{d}t$$

We use the points at n and $n+1$ to fit a linear polynomial to f_n, i.e. $f(t) = at + b$. At $t = 0$, $f_n = b$ and at $t = -\Delta t$, $f_{n-1} = -a\Delta t + b$. It follows that $b = f_n$ and $a = \frac{f_n - f_{n-1}}{\Delta t}$. Thus,

$$y_{n+1} = y_n + \int_0^{\Delta t} (at + b) \, \mathrm{d}t = y_n + \left(\frac{a}{2}(\Delta t)^2 + b\Delta t \right) = y_n + \left(\frac{3}{2} f_n - \frac{1}{2} f_{n-1} \right) \Delta t.$$

This defines the second order Adams-Bashforth method.

2.5.2 AB3

Similarly, fitting a second order $f(t) = at^2 + bt + c$ at times $t \in \{0, -\Delta t, -2\Delta t\}$ yields

$$c = f_n, \quad a = \frac{f_{n-2} - 2f_{n-1} + f_n}{2(\Delta t)^2}, \quad b = \frac{f_{n-2} - 4f_{n-1} + 3f_n}{2\Delta t}.$$

By integrating, we arrive at

$$y_{n+1} = y_n + \left(\frac{5}{12} f_{n-2} - \frac{16}{12} f_{n-1} + \frac{23}{12} f_n \right) \Delta t.$$

Higher order methods can be derived in a similar way. Note that AB1 is forward Euler. If we use f_{n+1} instead of f_{n-1}, we get the family of implicit Adams-Moulton methods (of which backward Euler is a member of).

Exercise 2.3
Show that the coefficients of f_i in an Adams-Bashforth method always sum to 1.

2.6 Numerical stability

Consider the differential equation $\frac{dy}{dt} = -\lambda y$ (for $\lambda > 0$) with initial condition $y(0) = y_0$. The exact solution is $y = y_0 e^{-\lambda t}$. We say that a scheme is *numerically stable* if $y_n \to 0$ as $n \to \infty$.

For forward Euler,

$$y_n = (1 - \lambda\Delta t)y_n = (1 - \lambda\Delta t)^n y_0.$$

Thus, if $|1 - \lambda\Delta t| > 1$, y_n grows and thus is unstable. So we need $\Delta t \leq \frac{2}{\lambda}$ for stability.

For backward Euler,

$$y_n = \frac{y_n}{1 + \lambda\Delta t} = \left(\frac{1}{1 + \lambda\Delta t}\right)^n y_0.$$

Since $\frac{1}{1+\lambda\Delta t} < 1$ for all $\Delta t > 0$, backward Euler is unconditionally stable. Similarly, implicit midpoint is also unconditionally stable as

$$y_n = \left(\frac{1 - \frac{\lambda}{2}\Delta t}{1 + \frac{\lambda}{2}\Delta t}\right)^n y_n.$$

3 Partial differential equations

A general linear second order PDE in two variables is given by

$$a\frac{\partial^2\varphi}{\partial x_1^2} + b\frac{\partial^2\varphi}{\partial x_1 \partial x_2} + c\frac{\partial^2\varphi}{\partial x_2^2} + d\frac{\partial\varphi}{\partial x_1} + e\frac{\partial\varphi}{\partial x_2} + f\varphi = g.$$

The discriminant is

$$\Delta = b^2 - 4ac \implies \begin{cases} \Delta < 0 & \text{elliptic,} \\ \Delta = 0 & \text{parabolic,} \\ \Delta > 0 & \text{hyperbolic.} \end{cases}$$

In general,

$$\sum_{ij} A_{ij}\frac{\partial^2\varphi}{\partial x_i \partial x_j} + \sum_i B_i\frac{\partial\varphi}{\partial x_i} + C\varphi = D.$$

This PDE is called *elliptic* if none of the eigenvalues vanish and all have the same sign; *parabolic* if ne eigenvalue vanishes and the remainder have the same sign; *hyperbolic* if none of the eigenvalues vanish and one has the opposite sign.

3.1 Parabolic PDEs

Consider the diffusion equation with diffusivity $\kappa > 0$

$$\frac{\partial\varphi}{\partial t} = \kappa\frac{\partial^2\varphi}{\partial x^2}.$$

This is a parabolic second order PDE.

Diffusing equations have causality, i.e. the direction of time matters. Information spreads on scales $\delta \sim \sqrt{\kappa t}$ as in Figure 6 - the time step cannot be larger, otherwise the scheme becomes unstable.

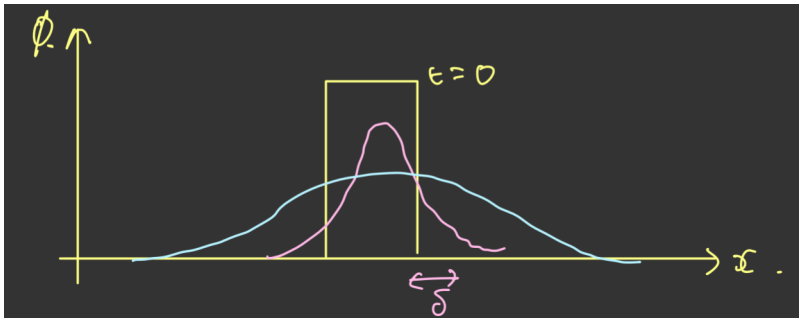

Figure 6: Diffusion spreading

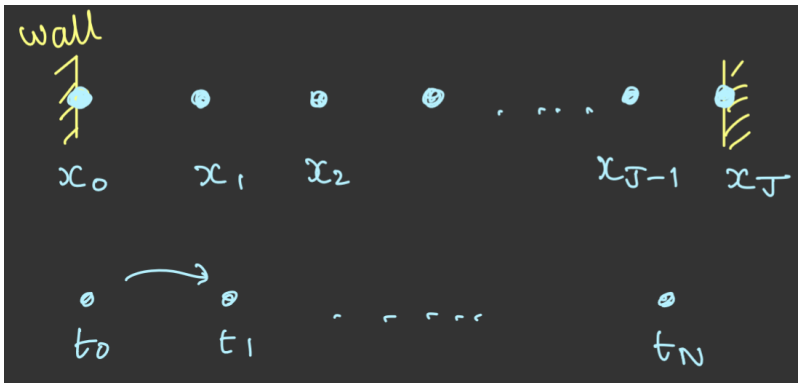

Figure 7: Discretising the diffusion equation domain

Example 3.1
Consider diffusion equation on $0 \leq x \leq L$ and $0 \leq t \leq T$. Let $\varphi_j^n = \varphi(x_j, t_n)$ where $t_n = n\Delta t$ and $x_j = j\Delta x$. The domain is discretised in Figure 7.

Using forward Euler,

$$\frac{\varphi_j^{n+1} - \varphi_j^n}{\Delta t} = \kappa \frac{\varphi_{j+1}^n - 2\varphi_j^n + \varphi_{j-1}^n}{(\Delta x)^2}.$$

It follows that

$$\varphi_j^{n+1} = \varphi_j^n + \frac{\kappa \Delta t}{(\Delta x)^2} \left(\varphi_{j+1}^n - 2\varphi_n^n + \varphi_{j-1}^n\right).$$

This is straightforward to solve as forward Euler explicit. On the other hand, implicit backward Euler is

$$\frac{\varphi_j^{n+1} - \varphi_j^n}{\Delta t} = \kappa \frac{\varphi_{j+1}^{n+1} - 2\varphi_j^{n+1} + \varphi_{j-1}^{n+1}}{(\Delta x)^2}.$$

There are three unknowns, so we can not solve this directly. We solve this in Section **??**.

3.2 Boundary conditions

A Dirichlet boundary condition is when we specify φ on solid boundaries. For example, $\varphi(0, t) = 1$ or $\varphi(L, t) = 0.5 \sin(2\pi t)$.

Neumann boundary conditions specify the normal derivative. For example, $\frac{\partial \varphi}{\partial x}(0, t) = 0$ (no flux), $\frac{\partial \varphi}{\partial x}(L, t) = F$ (specified flux).

We can mix the boundary conditions (Robin boundary conditions) to get $a\varphi + b\frac{\partial\varphi}{\partial x} = c$.

Dirichlet boundary conditions are easy to implement by setting φ at the boundary. Neumann boundary conditions are a bit more complicated. We can use a 1-sided derivative with a grid point on te boundary:

$$\frac{\partial\varphi}{\partial x}(0,t) = 0 \implies \frac{\varphi_1^n - \varphi_0^n}{\Delta x} = 0 \implies \varphi_0^n = \varphi_1^n.$$

This is only first order accurate. A better place for the boundary is between two midpoints, which results in a second order accurate centred derivative. This can be implemented in two ways. The first is using "ghost points", i.e. put φ_0 outside the domain. This may not be appropriate for complicated domains. The second way is to absorb the boundary conditions into the finite difference operator. For example, if $\kappa\frac{\partial\varphi}{\partial x}\big|_{0.5}^n = 0$,

$$\kappa\frac{\partial^2\varphi}{\partial x^2}\bigg|_1^n = \frac{\partial}{\partial x}\left(\kappa\frac{\partial\varphi}{\partial x}\right)\bigg|_1^n = \frac{\kappa\frac{\partial\varphi}{\partial x}\big|_{1.5}^n - \kappa\frac{\partial\varphi}{\partial x}\big|_{0.5}^n}{\Delta x} = \frac{\kappa}{\Delta x}(\varphi_2^n - \varphi_1^n).$$

4 Von Neumann analysis

5 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.