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

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

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

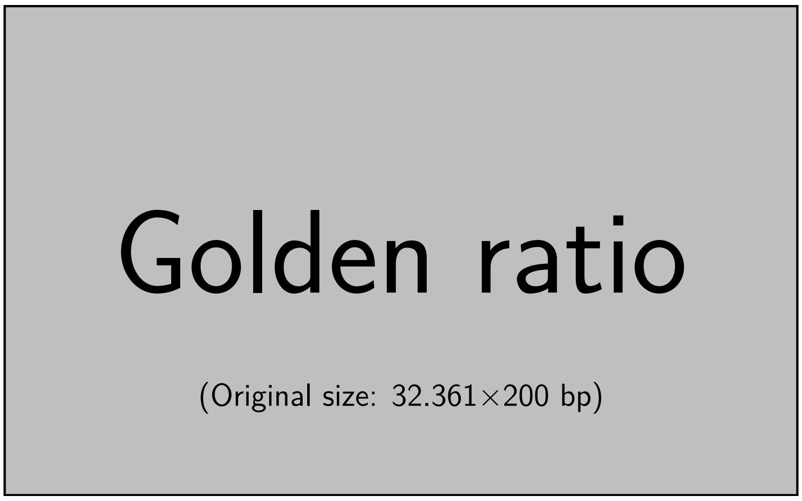

Figure 1: Bisection method

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.

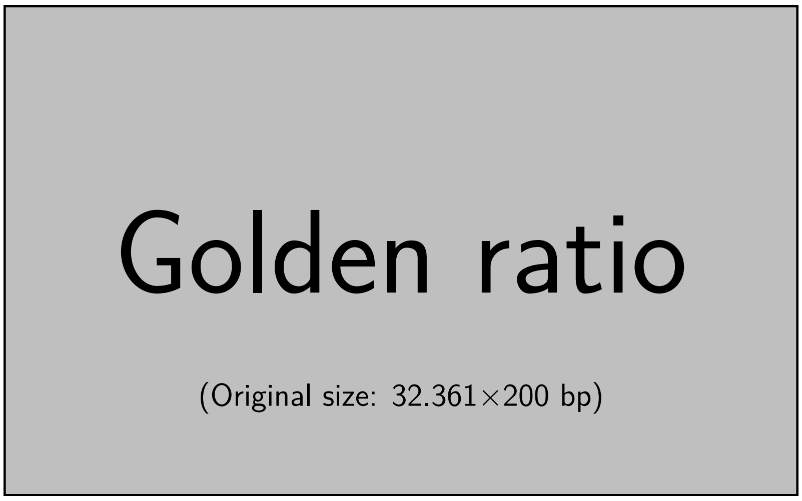

Figure 2: Newton's method

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 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

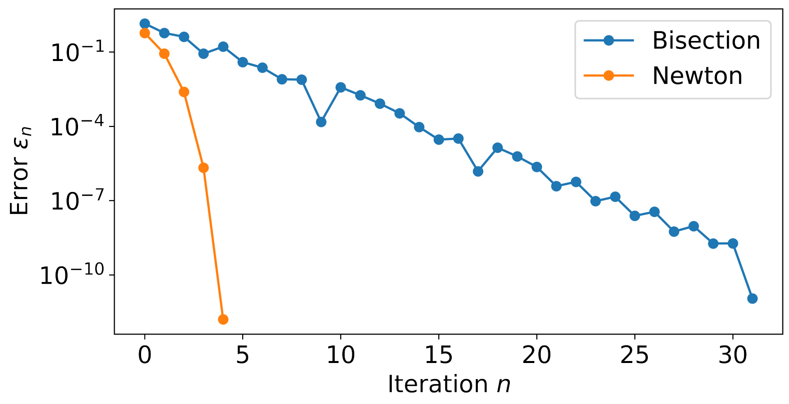

Figure 3: Bisection method vs Newton's method for approximating $\sqrt{2}$

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.

1.3 Higher dimensions

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

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

The iterative step of Newton's method becomes

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

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 7).

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

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

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

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

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.

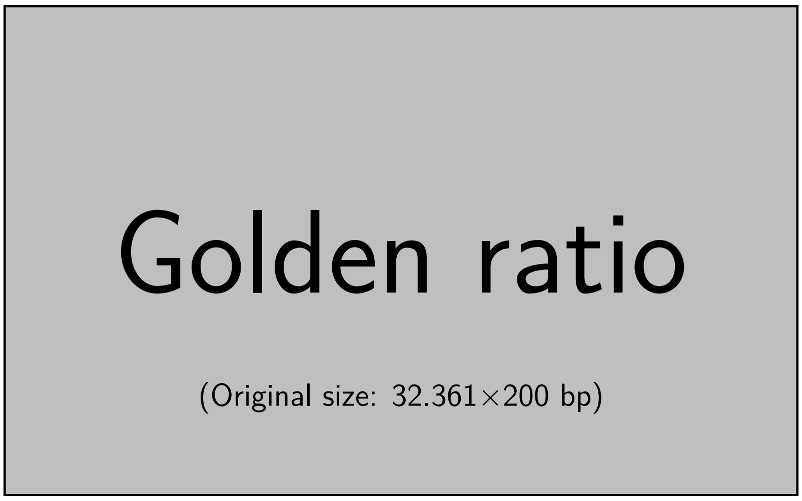

Figure 4: Euler

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.

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

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

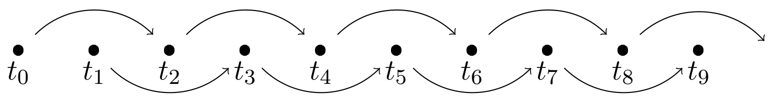

Figure 5: Leap-frog

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

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

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 $0.01 \le \nu \le 0.2$ 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

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

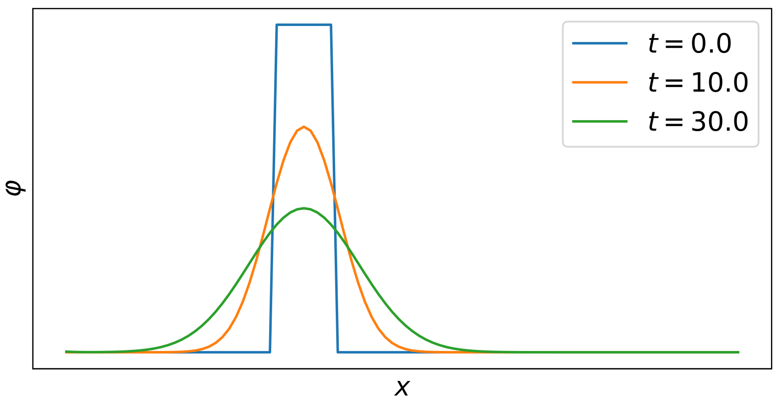

Figure 6: Diffusion with $\kappa = 1$ for the top hat function.

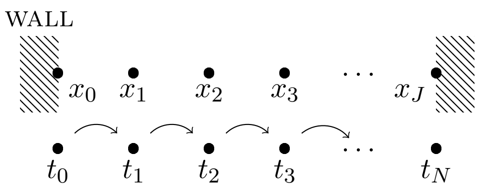

Figure 7: Discretising the diffusion equation domain

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.

Example 3.1
Consider diffusion equation on $0 \le x \le L$ and $0 \le t \le T$. 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

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

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 $\mathbf{n}\cdot\nabla\varphi$. 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 the 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 $\kappa \frac{\partial \varphi}{\partial x}\Big|_{0.5}^n = 0$,

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

3.3 Hyperbolic PDEs

Consider the wave equation

$$\frac{d^2 \varphi}{dt^2} = c^2 \frac{d^2 \varphi}{dx^2}.$$

This is a hyperbolic PDE. Factoring,

$$\left(\frac{\partial}{\partial t} + c\frac{\partial}{\partial x}\right)\left(\frac{\partial}{\partial t} - c\frac{\partial}{\partial x}\right)\varphi = 0. \tag{4}$$

Let $p = x + ct$ and $q = x - ct$, then $x = \frac{p+q}{2}$ and $t = \frac{p-q}{2c}$. From the chain rule, it follows that

$$\begin{aligned}
\frac{\partial \varphi}{\partial p} &= \frac{\partial \varphi}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial \varphi}{\partial t}\frac{\partial t}{\partial p} = \frac{1}{2}\frac{\partial \varphi}{\partial x} + \frac{1}{2c}\frac{\partial \varphi}{\partial t}, \\
\frac{\partial \varphi}{\partial q} &= \frac{\partial \varphi}{\partial x}\frac{\partial x}{\partial q} + \frac{\partial \varphi}{\partial t}\frac{\partial t}{\partial q} = \frac{1}{2}\frac{\partial \varphi}{\partial x} - \frac{1}{2c}\frac{\partial \varphi}{\partial t}. \\
\implies \frac{\partial}{\partial x} + c\frac{\partial}{\partial t} &= 2c\frac{\partial}{\partial p}, \quad \frac{\partial}{\partial x} - c\frac{\partial}{\partial t} = -2c\frac{\partial}{\partial q}.
\end{aligned}$$

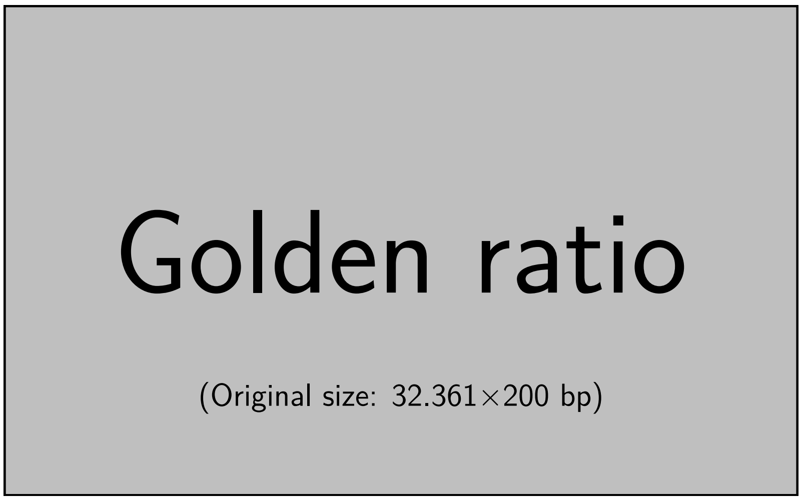

Figure 8: Waves moving at constant speed c

Substituting into Equation 4, we get the hyperbolic PDE

$$\frac{\partial^2 \varphi}{\partial p \partial q} = 0.$$

Integrating with respect to p and q,

$$\frac{\partial \varphi}{\partial q} = f'(q) \implies \varphi = f(q) + g(p),$$

for some choice of waves f and g. Thus $\varphi(x,t) = f(x-ct) + g(x+ct)$. We see that φ is constant along lines of $x - ct$ and $x + ct$ as in Figure 8.

Now consider the wave equation on $0 \leq t \leq \infty$ and $0 \leq x \leq L$. At time $t = 0$, since we have a second derivative in time, we need two initial conditions (corresponding to f and g). We need one boundary condition each on $x = 0$ (corresponding to f) and $x = L$ (corresponding to g). This is illustrated in Figure 9. For example, we could take boundary conditions $\varphi(x,0)$, $\frac{\partial}{\partial t}\varphi(x,0)$, $\varphi(0,t)$ and $\varphi(L,t)$.

Adding an advection term complicates things. One way to treat advection is to use sponge layers.

To solve the wave equation, we may use a centred difference for both second derivatives

$$U_j^{n+1} - 2U_j^n + U_j^{n-1}(\Delta t)^2 = c^2 \left(\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{\Delta x^2} \right),$$

where $U_j^n = \varphi(x_j t_n)$ This is second order in both time and space. For stability, we need $|\Delta x| < |c \Delta t|$, i.e. the wave shouldn't travel more than one grid space in one time step.

3.4 Advection equations

An advection equation has the form

$$\frac{\partial \varphi}{\partial t} + u \frac{\partial \varphi}{\partial x} = 0.$$

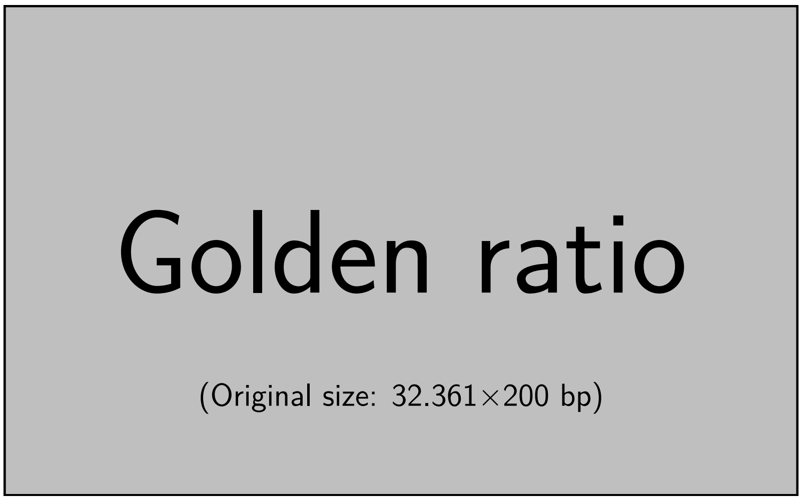

Figure 9: Wave domain discretise

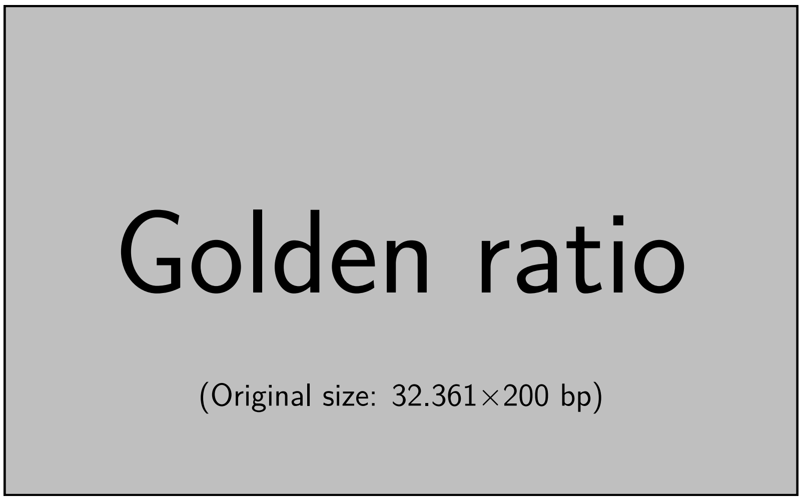

Figure 10: Advection Euler grid

This models a fluid parcel moving at velocity $u > 0$ while maintaining its φ. If u is constant, then $\varphi(x,t) = f(x - uy)$.

Using forward Euler, we get the "upwind" scheme

$$\frac{\varphi_j^{n+1} - \varphi_j^n}{\Delta t} + u\left(\frac{\varphi_j^n - \varphi_{j-1}^n}{\Delta x}\right) = 0.$$

This is illustrated in Figure 10. This is unstable if $u < 0$ or $u > \frac{\Delta x}{\Delta t}$ and is called the Courant–Friedrichs–Lewy (CFL) condition. If $u < 0$, swap the direction of the discretisation.

More generally,

$$\varphi_j^{n+1} = \varphi_j^n - \begin{cases} \frac{u\Delta t}{\Delta x}(\varphi_j^n - \varphi_{j-1}^n) & \text{if } u \geq 0, \\ \frac{u\Delta t}{\Delta x}(\varphi_{j+1}^n - \varphi_j^n) & \text{if } u < 0. \end{cases}$$

This is the general "upwind" scheme and is first order in space and time.

To get higher order in time, we can use an AB scheme. For second order space, we can try using a centred difference

$$\frac{\varphi_j^{n+1}-\varphi_j^n}{\Delta t}+u\left(\frac{\varphi_{j+1}^n-\varphi_{j-1}^n}{2\Delta x}\right)=0.$$

But this is an unstable scheme. However, the Lax-Friedrichs scheme works (and is stable) by adding diffusion in time

$$\frac{\varphi_j^{n+1}-\frac{1}{2}(\varphi_{j+1}^n+\varphi_{j-1}^n)}{\Delta t}+\frac{u(\varphi_{j+1}^n-\varphi_{j-1}^n)}{2\Delta x}=0.$$

3.4.1 Diffusive errors

Consider the upwind scheme

$$\varphi_{j-1}^n=\varphi_j^n-\left.\frac{\partial\varphi}{\partial x}\right|_j^n\Delta x+\left.\frac{\partial^2\varphi}{\partial x^2}\right|_j^n\frac{(\Delta x)^2}{2}+O((\Delta x)^3).$$

Then

$$\frac{\varphi_j^{n+1}-\varphi_j^n}{\Delta t}=-u\left(\frac{\varphi_j^n-\varphi_{j-1}^n}{\Delta x}\right)=-u\left.\frac{\partial\varphi}{\partial x}\right|_j^n+\frac{u\Delta x}{2}\left.\frac{\partial^2\varphi}{\partial x^2}\right|_j^n+O((\Delta x)^2).$$

The coefficient $\frac{u\Delta x}{2}$ is the effective diffusivity.

3.4.2 Dispersive errors

Consider centred advection. Using the Taylor expansion,

$$\varphi_{j\pm1}^n=\varphi_j^n\pm\left.\frac{\partial\varphi}{\partial x}\right|_j^n\Delta x+\left.\frac{\partial^2\varphi}{\partial x^2}\right|_j^n\frac{(\Delta x)^2}{2}\pm\left.\frac{\partial^3\varphi}{\partial x^3}\right|_j^n\frac{(\Delta x)^3}{6}+O((\Delta x)^4),$$

we get

$$u\frac{\partial\varphi}{\partial x}\approx u\left(\frac{\varphi_{j+1}^n-\varphi_{j-1}^n}{2\Delta x}\right)=u\left.\frac{\partial\varphi}{\partial x}\right|_j^n+\frac{u(\Delta x)^2}{6}\left.\frac{\partial^3\varphi}{\partial x^3}\right|_j^n.$$

Take the solution $\varphi=\varphi_0\exp(i(kx-wt))$. Pure advection becomes

$$\frac{\partial\varphi}{\partial t}+u\frac{\partial\varphi}{\partial x}=-\implies -i\omega+iku=0\implies\frac{\omega}{k}=\frac{d\omega}{dk}=u.$$

Numerically,

$$\frac{\partial\varphi}{\partial t}+u\frac{\partial\varphi}{\partial x}+\frac{u(\Delta x)^2}{6}\frac{\partial^3\varphi}{\partial x^3}=0\implies -i\omega+iku-ik^3\frac{u(\Delta x)^2}{6}=0$$

Then

$$\frac{\omega}{k}=u-\frac{u(\Delta x)^2}{6}k^2,\quad\frac{d\omega}{dk}=u-\frac{u(\Delta x)^2}{6}k^2.$$

4 Elliptic PDEs

A prototypical elliptic PDE is the Laplace equation

$$\nabla^2 \varphi = \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = 0.$$

Adding forcing, we get a Poisson equation

$$\nabla^2 \varphi = f(\mathbf{x}).$$

A Helmoltz equation is

$$\nabla^2 \varphi + k^2 \varphi = f(\mathbf{x}).$$

We need one boundary condition over entire boundary. If we use Neumann boundary conditions on the Poisson equation,

$$\iint \nabla^2 \varphi dA = \oint \nabla\varphi \cdot \, \mathrm{d}\mathbf{n} = \iint f(x) dA,$$

by the divergence theorem. So we must ensure the solution is consistent. With no forcing and zero Neumann boundary conditions, we can add any constant to φ, so we can either set $\iint \varphi dA = 0$ or $\varphi = 0$ at one grid point. This is important to prevent φ from drifting.

4.1 Numerical methods

Consider the 1D Poisson equation

$$\frac{d^2\varphi}{dx^2} = f(x)$$

on the domain $0 \le x \le L$ with boundary conditions $|phi(0) = a$ and $|phi(L) = b$. We may do centred difference on the second derivative

$$\frac{\varphi_{j+1} - 2\varphi_j + \varphi_{j-1}}{(\Delta x)^2} = f_j,$$

where $f_j = f(x_j)$. If we use 5 grid points ($j \in \{0, \dots, 4\}$), we have the system

$$\begin{aligned} \varphi_0 &= a \\ \varphi_0 - 2\varphi_1 + \varphi_2 &= f_1(\Delta x)^2 \\ \varphi_1 - 2\varphi_2 + \varphi_3 &= f_2(\Delta x)^2 \\ \varphi_2 - 2\varphi_3 + \varphi_4 &= f_3(\Delta x)^2 \\ \varphi_4 &= b \end{aligned}$$

This corresponds to the tri-diagonal matrix system

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \varphi_0 \\ \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \end{bmatrix} = \begin{bmatrix} a \\ f_1(\Delta x)^2 \\ f_2(\Delta x)^2 \\ f_3(\Delta x)^2 \\ b \end{bmatrix}$$

Note 4.1 (Useful commands)

- Python NumPy's `linalg.solve` solves a linear system.
- Matlab backslash.

We look at different matrix solvers in in 7.

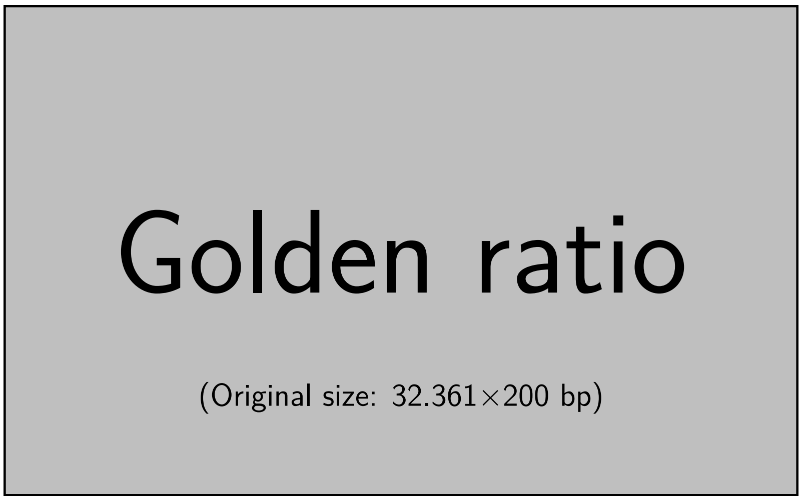

Figure 11: Periodic boundary conditions

4.2 Implicit time-stepping

Many implicit time stepping methods require solving elliptic equations, for example backwards Euler on a diffusion equation:

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

Rearranging, we get an elliptic equation

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

In matrix form,

$$\begin{bmatrix} 1 & & & & & \\ -\lambda & 1+2\lambda & -\lambda & & & \\ & -\lambda & 1+2\lambda & -\lambda & & \\ & & & \ddots & & \\ & & & -\lambda & 1+2\lambda & -\lambda \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} \varphi_0^{n+1} \\ \varphi_1^{n+1} \\ \varphi_2^{n+1} \\ \vdots \\ \varphi_{J-1}^{n+1} \\ \varphi_J^{n+1} \end{bmatrix} = \begin{bmatrix} \varphi_0^n \\ \varphi_1^n \\ \varphi_2^n \\ \vdots \\ \varphi_{J-1}^n \\ \varphi_J^n \end{bmatrix}$$

We are trading the benefit of no time step restriction at the expense of solving an elliptic problem.

4.3 Periodic boundary conditions

Sometimes, we need to use periodic boundary conditions (Figure 11); for example, longitude goes from 0° to 360°.

To implement, we can either duplicate ghost points as in Figure 12. This is easy, but can be error prone.

A better way is to modify the array indices at the periodic boundaries, for example,

$$\varphi_{J-1}^{n+1} = \varphi_{J-1}^n + \left(\frac{\kappa \Delta t}{(\Delta x)^2}\right)\left(\varphi_{J-2}^{n+1} - 2\varphi_{J-1}^{n+1} + \varphi_{j-1}^0\right).$$

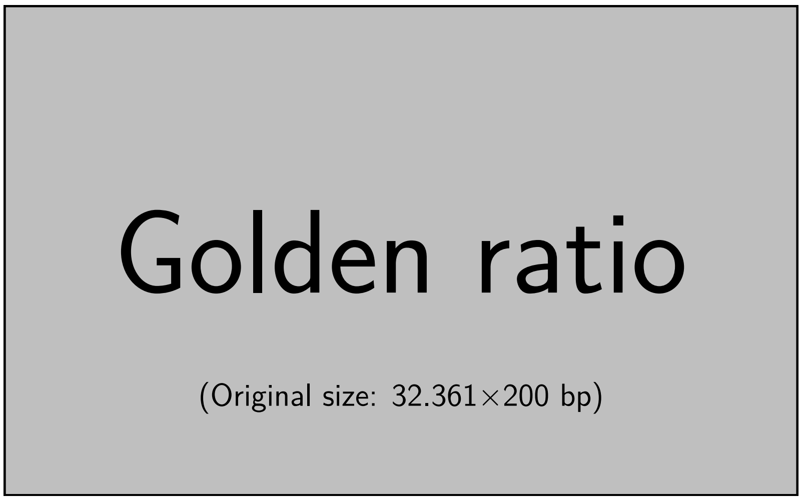

Figure 12: Periodic boundary conditions ghost points

4.4 Arakawa grid

$\bar{p}^y =$ is an average along the y direction.

5 Von Neumann analysis

Von Neumann analysis studies the stability of numerical methods by treating the numerical solution as a series of waves. If any of these waves grow, the scheme is unstable.

Example 5.1 (Diffusion equation with forward Euler)
The forward Euler discretisation of $\frac{\partial \varphi}{\partial t} = \kappa \frac{\partial^2 \varphi}{\partial x^2}$ ($\kappa > 0$ and $\Delta t > 0$) is

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

Suppose we have a solution $\varphi_j^n = A^{(n)}(k) \exp(ikx_j)$. Substituting this into the discretisation, ...

Using the fact that $x_j = j\Delta x$, $\exp(ikx_{j+1}) = \exp(ikx_j)\exp(ik\Delta x)$. Thus...

It follows that

$$\frac{A^{(n+1)}}{A^{(n)}} = 1 - 2\frac{\kappa \Delta t}{(\Delta x)^2}(1 - \cos(k\Delta x)).$$

For the scheme to be stable, we want $\left|\frac{A^{(n+1)}}{A^{(n)}}\right| \leq 1$. Since $0 \leq 1 - \cos \leq 2$, we have that

$$\left|\frac{A^{(n+1)}}{A^{(n)}}\right| \implies \frac{2\kappa \Delta t}{(\Delta x)^2} \leq 1.$$

Thus, for stability, we need $\Delta t \leq \frac{(\Delta x)^2}{2\kappa}$.

Example 5.2 (Diffusion equation with backward Euler)
The backward Euler discretisation is

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

The Von Neumann analysis is identical to forward Euler, except we have $A^{(n+1)}$ instead of $A^{(n)}$, i.e.

$$\frac{A^{(n+1)}-A^{(n)}}{\Delta t}=\frac{\kappa}{(\Delta x)^2}A^{(n+1)}(2\cos(k\Delta x)-2).$$

It follows that

$$\frac{A^{(n+1)}}{A^{(n)}}=...\le 1.$$

Thus backward Euler is stable for any choice of Δt.

Example 5.3 (Upwind advection with forward Euler)
Forward Euler for ... $(u>0)$ is

$$\frac{\varphi_j^{n+1}-\varphi_j^n}{\Delta t}+u\frac{\varphi_j^n-\varphi_{j-1}^n}{\Delta x}=0.$$

Substituting in $\varphi_j^n=A^{(n)}(k)\exp(ikx_j)$,

$$\frac{A^{(n+1)}+A^{(n)}}{\Delta t}+...=0.$$

It follows that

$$\frac{A^{(n+1)}}{A^{(n)}}=1-\frac{u\Delta t}{\Delta x}(1-\exp(-ik\Delta x)).$$

Taking norms,

$$\begin{aligned}\left|\frac{A^{(n+1)}}{A^{(n)}}\right|^2&=\left(1-\frac{u\Delta t}{\Delta x}(1-\exp(-ik\Delta x))\right)\left(1-\frac{u\Delta t}{\Delta x}(1-\exp(ik\Delta x))\right)\\&=1-\frac{u\Delta t}{\Delta x}(2-\exp(ik\Delta x)-\exp(-ik\Delta x))+\left(\frac{u\Delta t}{\Delta x}\right)^2(2-\exp(ik\Delta x)-\exp(-ik\Delta x))\\&=1-\frac{u\Delta t}{\Delta x}\left(1-\frac{u\Delta t}{\Delta x}\right)(2-2\cos(k\Delta x)).\end{aligned}$$

Since $2-2\cos\ge 0$, for instability, we need

$$\left|[\frac{A^{(n+1)}}{A^{(n)}}\right|^2\ge 1 \text{ if } \frac{u\Delta t}{\Delta x}>1 \text{ or } \frac{u\Delta t}{\Delta x}<0.$$

It follows that downwind is unstable. So we have stability if upwind and $\Delta t<\frac{\Delta x}{u}$.

Example 5.4 (FTCS)

$$\frac{\varphi_j^{n+1}-\varphi_j^n}{\Delta t}+u\frac{\varphi_{j+1}^n-\varphi_{j-1}^n}{2\Delta x}=0.$$

Substituting $\varphi_j^n = A^{(n)}(k)\exp(ikx_j)$,

$$\frac{A^{(n+1)}}{A^{(n)}} - 1 + \frac{iu\Delta t}{\Delta x}\left(\frac{\exp(ik\Delta x) - \exp(-ik\Delta x)}{2i}\right) = 0.$$

It follows that

$$\left|\frac{A^{(n+1)}}{A^{(n)}}\right|^2 = \left|1 - \frac{iu\Delta t}{\Delta x}\sin(k\Delta x)\right| = 1 + \left(\frac{u\Delta t}{\Delta x}\right)^2 \sin^2(k\Delta x) \geq 1.$$

This scheme is always unstable.

6 Other numerical methods

7 Numerical linear algebra

Matrix equations turn up in many places, for example when solving an elliptic equation. In general, we wish to solve

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix for n very large. It is almost always inefficient to invert A, which takes $O(n^3)$ time and can be error prone, as there are faster methods that are $O(n^i)$ for $i < 3$. There are two main classes of methods, direct solvers and iterative solvers.

7.1 Gaussian elimination

The usual method of solving N equations in N unknowns is to write one variable in terms of the others, then to eliminate that variable from the remaining $N - 1$ equations. After proceeding inductively, the N-th equation will have one variable, and then we can back substitute to find the other variables.

Example 7.1
Consider the linear system

$$\begin{cases} 2x_1 + x_2 - x_3 = 8 \\ -3x_1 - x_2 + 2x_3 = -11 \\ -2x_1 + x_2 + 2x_3 = -3 \end{cases}.$$

The augmented matrix form is

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right].$$

To reduce this matrix, we can perform a sequence of elementary row operations (which are scaling rows, swapping rows, and adding a multiple of one row to another).

We wish to first reduce the augmented matrix to an upper diagonal matrix of the form

$$\left[\begin{array}{ccc|c} \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot \end{array}\right].$$

Then, we can perform back substitution to get a diagonal matrix

$$\left[\begin{array}{ccc|c} \cdot & 0 & 0 & \cdot \\ 0 & \cdot & 0 & \cdot \\ 0 & 0 & \cdot & \cdot \end{array}\right].$$

For our example, the sequence of row reductions to get an upper triangular matrix is

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right].$$

Back substitution gets us

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 1 & 0 & 7 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 0 & 0 & 4 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array}\right].$$

Rescaling,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right] \sim \left[\begin{array}{ccc|c} 2 & 0 & 0 & 4 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array}\right] \sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right].$$

It follows that $x_1 = 2$, $x_2 = 3$, $x_3 = -1$.

The general algorithm is [insert algorithm here]

The first stage is $O(n^3)$, and the back substitution is $O(n^2)$. This is bad!

7.2 LU decomposition

The goal of LU decomposition is to factor $A = LU$, where L is lower triangular and U is upper triangular. After factoring, we can forward substitute and solve $LU\mathbf{x} = \mathbf{b}$ for $U\mathbf{x} = L^{-1}\mathbf{b}$, then to back substitute to solve for $\mathbf{x}$. Note that LU decomposition is not unique, but insisting that the diagonal entries of L are 1 enforces uniqueness.

Example 7.2
For a 3×3 matrix,

$$\begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix}.$$

By equating entries,

$$A = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{3}{2} & 1 & 0 \\ -1 & 4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & -1 \end{bmatrix} = LU.$$

It follows that via forward and backward substitution that

$$U\mathbf{x} = L^{-1}\mathbf{b} = \begin{bmatrix} 8 \\ 1 \\ 1 \end{bmatrix} \implies \mathbf{x} = U^{-1}L^{-1}b = \begin{bmatrix} 2 \\ 3 \\ -1 \end{bmatrix}.$$

The general algorithm for LU is [insert algorithm]

The most expensive step of this algorithm is the LU factorisation. When solving a PDE, we need to keep solving the matrix system with the same matrix A, so it is worth computing $A = LU$ once, and reusing the factorisation for each time step and performing cheap substitution.

If a diagonal entry is zero, we can swap rows to avoid division by zero. To minimise errors, use the row with the largest pivot. This is called LU factorisation with (partial) pivoting.

Matrices from finite difference are sparse. LU decomposition is slow and takes a lot of memory (for very large matrices). Iterative methods are better for large sparse matrices.

7.3 Jacobi

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.