Unit 3: Numerical Calculus (Part 2)

Integration of ODE Initial Value Problems

In this chapter we consider problems of the form

$$y'(t) = f(t, y), \quad y(0) = y_0$$

Here $y(t) \in \mathbb{R}^n$ and $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$

Writing this system out in full, we have:

$$y'(t) = \begin{bmatrix} y_1'(t) \\ y_2'(t) \\ \vdots \\ y_n'(t) \end{bmatrix} = \begin{bmatrix} f_1(t, y) \\ f_2(t, y) \\ \vdots \\ f_n(t, y) \end{bmatrix} = f(t, y(t))$$

This is a system of n coupled ODEs for the variables $y_1, y_2, \ldots, y_n$

ODE IVPs

Initial Value Problem implies that we know $y(0)$, *i.e.*
$y(0) = y_0 \in \mathbb{R}^n$ is the initial condition

The order of an ODE is the highest-order derivative that appears

Hence $y'(t) = f(t, y)$ is a first order ODE system

ODE IVPs

We only consider first order ODEs since higher order problems can be transformed to first order by introducing extra variables

For example, recall Newton's Second Law:

$$y''(t) = \frac{F(t, y, y')}{m}, \qquad y(0) = y_0, y'(0) = v_0$$

Let $v = y'$, then

$$\begin{aligned} v'(t) &= \frac{F(t, y, v)}{m} \\ y'(t) &= v(t) \end{aligned}$$

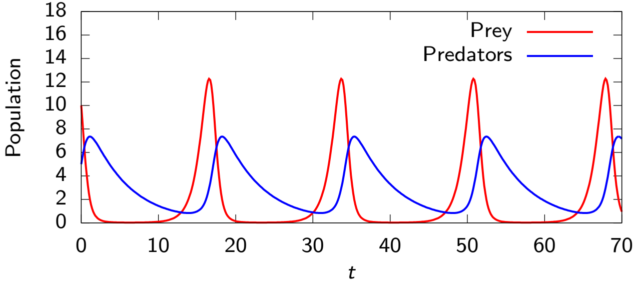
and $y(0) = y_0$, $v(0) = v_0$

ODE IVPs: A Predator–Prey ODE Model

[*l-v.py*] For example, a two-variable nonlinear ODE, the Lotka–Volterra equation, can be used to model populations of two species:

$$y' = \begin{bmatrix} y_1(\alpha_1 - \beta_1 y_2) \\ y_2(-\alpha_2 + \beta_2 y_1) \end{bmatrix} \equiv f(y)$$

The α and β are modeling parameters, describe birth rates, death rates, predator-prey interactions



ODEs in Python and MATLAB

Both Python and MATLAB have very good ODE IVP solvers

They employ adaptive time-stepping (h is varied during the calculation) to increase efficiency

Python has functions `odeint` (a general purpose routine) and `ode` (a routine with more options)

Most popular MATLAB function is `ode45`, which uses the classical fourth-order Runge–Kutta method

In the remainder of this chapter we will discuss the properties of methods like the Runge–Kutta method

Approximating an ODE IVP

Given $y' = f(t, y)$, $y(0) = y_0$: suppose we want to approximate y at $t_k = kh$, $k = 1, 2, \ldots$

Notation: Let y_k be our approx. to $y(t_k)$

Euler's method: Use finite difference approx. for y' and sample $f(t, y)$ at t_k:[1]

$$\frac{y_{k+1} - y_k}{h} = f(t_k, y_k)$$

Note that this, and all methods considered in this chapter, are written the same regardless of whether y is a vector or a scalar

[1] Note that we replace $y(t_k)$ by y_k

Euler's Method

Quadrature-based interpretation: integrating the ODE $y' = f(t, y)$ from t_k to t_{k+1} gives

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

Apply $n = 0$ Newton–Cotes quadrature to $\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$, based on interpolation point t_k:

$$\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s \approx (t_{k+1} - t_k) f(t_k, y_k) = hf(t_k, y_k)$$

Again, this gives Euler's method:

$$y_{k+1} = y_k + hf(t_k, y_k)$$

Python example: [*euler.py*] Euler's method for $y' = \lambda y$

Backward Euler Method

We can derive other methods using the same quadrature-based approach

Apply $n = 0$ Newton–Cotes quadrature based on interpolation point t_{k+1} to

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

to get the backward Euler method:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$$

Backward Euler Method

(Forward) Euler method is an explicit method: we have an explicit formula for y_{k+1} in terms of y_k

$$y_{k+1} = y_k + hf(t_k, y_k)$$

Backward Euler is an implicit method, we have to solve for y_{k+1} which requires some extra work

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1})$$

Backward Euler Method

For example, approximate $y' = 2\sin(ty)$ using backward Euler:

At the first step ($k = 1$), we get

$$y_1 = y_0 + h\sin(t_1 y_1)$$

To compute y_1, let $F(y_1) \equiv y_1 - y_0 - h\sin(t_1 y_1)$ and solve for $F(y_1) = 0$ via, say, Newton's method

Hence implicit methods are more complicated and more computationally expensive at each time step

Why bother with implicit methods? We'll see why shortly . . .

Trapezoid Method

We can derive methods based on higher-order quadrature

Apply $n = 1$ Newton–Cotes quadrature (Trapezoid rule) at t_k, t_{k+1} to

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

to get the Trapezoid Method:

$$y_{k+1} = y_k + \frac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_{k+1})\right)$$

One-Step Methods

The three methods we've considered so far have the form

$$\begin{aligned} y_{k+1} &= y_k + h\Phi(t_k, y_k; h) && \text{(explicit)} \\ y_{k+1} &= y_k + h\Phi(t_{k+1}, y_{k+1}; h) && \text{(implicit)} \\ y_{k+1} &= y_k + h\Phi(t_k, y_k, t_{k+1}, y_{k+1}; h) && \text{(implicit)} \end{aligned}$$

where the choice of the function Φ determines our method

These are called one-step methods: y_{k+1} depends on y_k

(One can also consider multistep methods, where y_{k+1} depends on earlier values $y_{k-1}, y_{k-2}, \ldots$; we'll discuss this briefly later)

Convergence

We now consider whether one-step methods converge to the exact solution as $h \to 0$

Convergence is a crucial property, we want to be able to satisfy an accuracy tolerance by taking h sufficiently small

In general a method that isn't convergent will give misleading results and is useless in practice!

Convergence

We define global error, e_k, as the total accumulated error at $t = t_k$

$$e_k \equiv y(t_k) - y_k$$

We define truncation error, T_k, as the amount “left over” at step k when we apply our method to the exact solution and divide by h

e.g. for an explicit one-step ODE approximation, we have

$$T_k \equiv \frac{y(t_{k+1}) - y(t_k)}{h} - \Phi(t_k, y(t_k); h)$$

Convergence

The truncation error defined above determines the local error introduced by the ODE approximation

For example, suppose $y_k = y(t_k)$, then for the case above we have

$$hT_k \equiv y(t_{k+1}) - y_k - h\Phi(t_k, y_k; h) = y(t_{k+1}) - y_{k+1}$$

Hence hT_k is the error introduced in one step of our ODE approximation[2]

Therefore the global error e_k is determined by the accumulation of the T_j for $j = 0, 1, \dots, k-1$

Now let's consider the global error of the Euler method in detail

[2]Because of this fact, the truncation error is defined as hT_k in some texts

Convergence

Theorem: Suppose we apply Euler's method for steps $1, 2, \ldots, M$, to $y' = f(t, y)$, where f satisfies a Lipschitz condition:

$$|f(t,u) - f(t,v)| \leq L_f |u - v|,$$

where $L_f \in \mathbb{R}_{>0}$ is called a Lipschitz constant. Then

$$|e_k| \leq \frac{\left(e^{L_f t_k} - 1\right)}{L_f} \left[\max_{0 \leq j \leq k-1} |T_j|\right], k = 0, 1, \ldots, M,$$

where T_j is the Euler method truncation error.[3]

[3]Notation used here supposes that $y \in \mathbb{R}$, but the result generalizes naturally to $y \in \mathbb{R}^n$ for $n > 1$

Convergence

Proof: From the definition of truncation error for Euler's method we have

$$y(t_{k+1}) = y(t_k) + hf(t_k, y(t_k); h) + hT_k$$

Subtracting $y_{k+1} = y_k + hf(t_k, y_k; h)$ gives

$$e_{k+1} = e_k + h\left[f(t_k, y(t_k)) - f(t_k, y_k)\right] + hT_k,$$

hence

$$|e_{k+1}| \leq |e_k| + hL_f|e_k| + h|T_k| = (1 + hL_f)|e_k| + h|T_k|$$

Convergence

Proof (continued . . .):

This gives a geometric progression, *e.g.* for $k = 2$ we have

$$\begin{aligned} |e_3| &\leq (1 + hL_f)|e_2| + h|T_2| \\ &\leq (1 + hL_f)((1 + hL_f)|e_1| + h|T_1|) + h|T_2| \\ &\leq (1 + hL_f)^2 h|T_0| + (1 + hL_f)h|T_1| + h|T_2| \\ &\leq h\left[\max_{0\leq j\leq 2} |T_j|\right] \sum_{j=0}^{2} (1 + hL_f)^j \end{aligned}$$

Or, in general

$$|e_k| \leq h\left[\max_{0\leq j\leq k-1} |T_j|\right] \sum_{j=0}^{k-1} (1 + hL_f)^j$$

Convergence

Proof (continued . . .):

Hence use the formula

$$\sum_{j=0}^{k-1} r^j = \frac{1 - r^k}{1 - r}$$

with $r \equiv (1 + hL_f)$, to get

$$|e_k| \leq \frac{1}{L_f} \left[\max_{0 \leq j \leq k-1} |T_j| \right] \left((1 + hL_f)^k - 1 \right)$$

Finally, we use the bound[4] $1 + hL_f \leq \exp(hL_f)$ to get the desired result. $\square$

[4]For $x \geq 0$, $1 + x \leq \exp(x)$ by power series expansion $1 + x + x^2/2 + \cdots$

Convergence: Lipschitz Condition

A simple case where we can calculate a Lipschitz constant is if $y \in \mathbb{R}$ and f is continuously differentiable

Then from the mean value theorem we have:

$$|f(t,u) - f(t,v)| = |f_y(t,\theta)||u - v|,$$

for $\theta \in (u, v)$

Hence we can set:

$$L_f = \max_{\substack{t \in [0, t_M] \\ \theta \in (u,v)}} |f_y(t,\theta)|$$

Convergence: Lipschitz Condition

However, f doesn't have to be continuously differentiable to satisfy Lipschitz condition!

e.g. let $f(x) = |x|$, then $|f(x) - f(y)| = ||x| - |y|| \leq |x - y|$,[5] hence $L_f = 1$ in this case

[5] This is the reverse triangle inequality

Convergence

For a fixed t (*i.e.* $t = kh$, as $h \to 0$ and $k \to \infty$), the factor $(e^{L_f t} - 1)/L_f$ in the bound is a constant

Hence the global convergence rate for each fixed t is given by the dependence of T_k on h

Our proof was for Euler's method, but the same dependence of global error on local error holds in general

We say that a method has order of accuracy p if $|T_k| = O(h^p)$ (where p is an integer)

Hence ODE methods with order ≥ 1 are convergent

Order of Accuracy

Forward Euler is first order accurate:

$$\begin{aligned}
T_k &\equiv \frac{y(t_{k+1}) - y(t_k)}{h} - f(t_k, y(t_k)) \\
&= \frac{y(t_{k+1}) - y(t_k)}{h} - y'(t_k) \\
&= \frac{y(t_k) + hy'(t_k) + h^2 y''(\theta)/2 - y(t_k)}{h} - y'(t_k) \\
&= \frac{h}{2} y''(\theta)
\end{aligned}$$

Order of Accuracy

Backward Euler is first order accurate:

$$\begin{aligned}
T_k &\equiv \frac{y(t_{k+1}) - y(t_k)}{h} - f(t_{k+1}, y(t_{k+1})) \\
&= \frac{y(t_{k+1}) - y(t_k)}{h} - y'(t_{k+1}) \\
&= \frac{y(t_{k+1}) - y(t_{k+1}) + hy'(t_{k+1}) - h^2 y''(\theta)/2}{h} - y'(t_{k+1}) \\
&= -\frac{h}{2} y''(\theta)
\end{aligned}$$

Order of Accuracy

Trapezoid method is second order accurate:

Let's prove this using a quadrature error bound, recall that:

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

and hence

$$\frac{y(t_{k+1}) - y(t_k)}{h} = \frac{1}{h}\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s$$

So

$$T_k = \frac{1}{h}\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s - \frac{1}{2}\left[f(t_k, y(t_k)) + f(t_{k+1}, y(t_{k+1}))\right]$$

Order of Accuracy

Hence

$$\begin{aligned} T_k &= \frac{1}{h}\left[\int_{t_k}^{t_{k+1}} f(s, y(s))\mathrm{d}s - \frac{h}{2}\left(f(t_k, y(t_k)) + f(t_{k+1}, y(t_{k+1}))\right)\right] \\ &= \frac{1}{h}\left[\int_{t_k}^{t_{k+1}} y'(s)\mathrm{d}s - \frac{h}{2}\left(y'(t_k) + y'(t_{k+1})\right)\right] \end{aligned}$$

Therefore T_k is determined by the trapezoid rule error for the integrand y' on $t \in [t_k, t_{k+1}]$

Recall that trapezoid quadrature rule error bound depended on $(b-a)^3 = (t_{k+1} - t_k)^3 = h^3$ and hence

$$T_k = O(h^2)$$

Order of Accuracy

The table below shows global error at $t = 1$ for $y' = y$, $y(0) = 1$ for (forward) Euler and trapezoid

h	E_{Euler}	E_{Trap}
2.0e-2	2.67e-2	9.06e-05
1.0e-2	1.35e-2	2.26e-05
5.0e-3	6.76e-3	5.66e-06
2.5e-3	3.39e-3	1.41e-06

$$h \to h/2 \implies E_{\text{Euler}} \to E_{\text{Euler}}/2$$

$$h \to h/2 \implies E_{\text{Trap}} \to E_{\text{Trap}}/4$$

Stability

So far we have discussed convergence of numerical methods for ODE IVPs, *i.e.* asymptotic behavior as $h \to 0$

It is also crucial to consider stability of numerical methods: for what (finite and practical) values of h is our method stable?

We want our method to be well-behaved for as large a step size as possible

All else being equal, larger step sizes $\implies$ fewer time steps $\implies$ more efficient!

Stability

In this context, the key idea is that we want our methods to inherit the stability properties of the ODE

If an ODE is unstable, then we can't expect our discretization to be stable

But if an ODE is stable, we want our discretization to be stable as well

Hence we first discuss ODE stability, independent of numerical discretization

ODE Stability

Consider an ODE $y' = f(t, y)$, and

- Let $y(t)$ be the solution for initial condition $y(0) = y_0$
- Let $\hat{y}(t)$ be the solution for initial condition $\hat{y}(0) = \hat{y}_0$

The ODE is stable if:

For every $\epsilon > 0$, $\exists \delta > 0$ such that

$$\|\hat{y}_0 - y_0\| \leq \delta \implies \|\hat{y}(t) - y(t)\| \leq \epsilon$$

for all $t \geq 0$

"Small input perturbation leads to small perturbation in the solution"

ODE Stability

Stronger form of stability, asymptotic stability: $\|\hat{y}(t) - y(t)\| \to 0$ as $t \to \infty$, perturbations decay over time

These two definitions of stability are properties of the ODE, independent of any numerical algorithm

This nomenclature is a bit confusing compared to previous Units:

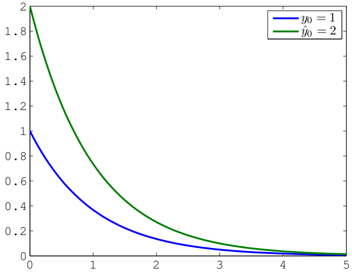
- We previously referred to this type of property as the conditioning of the problem
- Stability previously referred only to properties of a numerical approximation

In ODEs (and PDEs), it is standard to use stability to refer to sensitivity of both the mathematical problem and numerical approx.

ODE Stability

Consider stability of $y' = \lambda y$ (assuming $y(t) \in \mathbb{R}$) for different values of λ

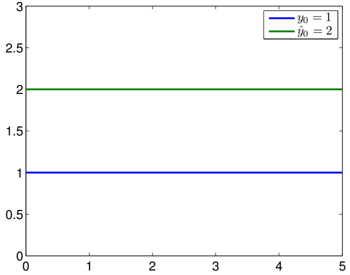
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = -1$, asymptotically stable

ODE Stability

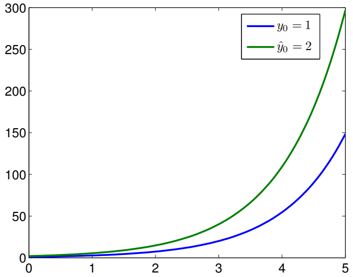
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = 0$, stable

ODE Stability

$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = 1$, unstable

ODE Stability

More generally, we can allow λ to be a complex number: $\lambda = a + ib$

Then $y(t) = y_0 e^{(a+ib)t} = y_0 e^{at} e^{ibt} = y_0 e^{at}(\cos(bt) + i\sin(bt))$

The key issue for stability is now the sign of $a = \mathrm{Re}(\lambda)$:

- $\mathrm{Re}(\lambda) < 0 \implies$ asymptotically stable
- $\mathrm{Re}(\lambda) = 0 \implies$ stable
- $\mathrm{Re}(\lambda) > 0 \implies$ unstable

ODE Stability

Our understanding of the stability of $y' = \lambda y$ extends directly to the case $y' = Ay$, where $y \in \mathbb{R}^n, A \in \mathbb{R}^{n\times n}$

Suppose that A is diagonalizable, so that we have the eigenvalue decomposition $A = V\Lambda V^{-1}$, where

- $\Lambda = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$, where the λ_j are eigenvalues
- V is matrix with eigenvectors as columns, $v_1, v_2, \ldots, v_n$

Then,

$$y' = Ay = V\Lambda V^{-1}y \implies V^{-1}y' = \Lambda V^{-1}y \implies z' = \Lambda z$$

where $z \equiv V^{-1}y$ and $z_0 \equiv V^{-1}y_0$

ODE Stability

Hence we have n decoupled ODEs for z, and stability of z_i is determined by λ_i for each i

Since z and y are related by the matrix V, then (roughly speaking) if all z_i are stable then all y_i will also be stable[6]

Hence assuming that V is well-conditioned, then we have:
$\text{Re}(\lambda_i) \leq 0$ for $i = 1, \ldots, n \implies y' = Ay$ is a stable ODE

Next we consider stability of numerical approximations to ODEs

[6] "Roughly speaking" here because V can be ill-conditioned — a more precise statement is based on "pseudospectra", outside the scope of AM205

ODE Stability

Numerical approximation to an ODE is stable if:

$$\boxed{\begin{array}{l}\text{For every } \epsilon > 0,\ \exists \delta > 0 \text{ such that} \\[1em] \qquad\qquad \|\hat{y}_0 - y_0\| \leq \delta \implies \|\hat{y}_k - y_k\| \leq \epsilon \\[1em] \text{for all } k \geq 0\end{array}}$$

Key idea: We want to develop numerical methods that mimic the stability properties of the exact solution

That is, if the ODE we're approximating is unstable, we can't expect the numerical approximation to be stable!

Stability

Since ODE stability is problem dependent, we need a standard "test problem" to consider

The standard test problem is the simple scalar ODE $y' = \lambda y$

Experience shows that the behavior of a discretization on this test problem gives a lot of insight into behavior in general

Ideally, to reproduce stability of the ODE $y' = \lambda y$, we want our discretization to be stable for all $\text{Re}(\lambda) \leq 0$

Stability: Forward Euler

Consider forward Euler discretization of $y' = \lambda y$:

$$y_{k+1} = y_k + h\lambda y_k = (1 + h\lambda) y_k \implies y_k = (1 + h\lambda)^k y_0$$

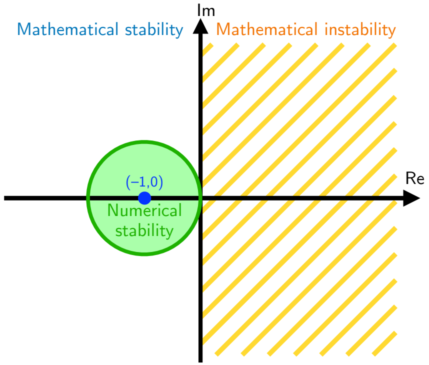
Here $1 + h\lambda$ is called the amplification factor

Hence for stability, we require $|1 + \bar{h}| \leq 1$, where $\bar{h} \equiv h\lambda$

Let $\bar{h} = a + ib$, then $|1 + a + ib|^2 \leq 1^2 \implies (1 + a)^2 + b^2 \leq 1$

Stability: Forward Euler

Hence forward Euler is stable if $\bar{h} \in \mathbb{C}$ is inside the disc with radius 1, center $(-1, 0)$: This is a subset of "left-half plane," $\text{Re}(\bar{h}) \leq 0$



As a result we say that the forward Euler method is conditionally stable: when $\text{Re}(\lambda) \leq 0$ we have to restrict h to ensure stability

Stability: Forward Euler

For example, given $\lambda \in \mathbb{R}_{<0}$, we require

$$-2 \leq h\lambda \leq 0 \implies h \leq -2/\lambda$$

Hence "larger negative λ" implies tighter restriction on h:

$$\begin{aligned} \lambda = -10 \quad &\implies \quad h \leq 0.2 \\ \lambda = -200 \quad &\implies \quad h \leq 0.01 \end{aligned}$$

Python example: [*e_stab.py*] Stability of the forward Euler method

Stability: Backward Euler

In comparison, consider backward Euler discretization for $y' = \lambda y$:

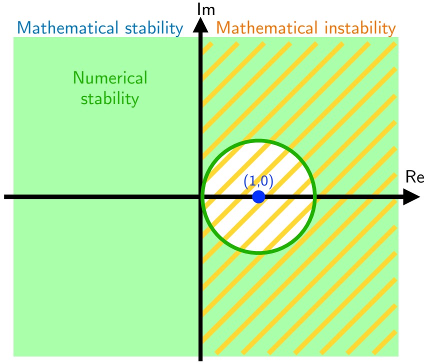
$$y_{k+1} = y_k + h\lambda y_{k+1} \implies y_k = \left(\frac{1}{1 - h\lambda}\right)^k y_0$$

Here the amplification factor is $\frac{1}{1-h\lambda}$

Hence for stability, we require $\frac{1}{|1-h\lambda|} \leq 1$

Stability: Backward Euler

Again, let $\bar{h} \equiv h\lambda = a + ib$, we need $1^2 \le |1 - (a+ib)|^2$, *i.e.* $(1-a)^2 + b^2 \ge 1$



Hence, for $\mathrm{Re}(\lambda) \le 0$, this is satisfied for any $h > 0$

As a result we say that the backward Euler method is unconditionally stable: no restriction on h for stability

Stability

Implicit methods generally have larger stability regions than explicit methods! Hence we can take larger timesteps with implicit

But explicit methods require less work per time-step since they don't need to solve for y_{k+1}

Therefore there is a tradeoff: The choice of method should depend on the details of the problem at hand

Runge–Kutta Methods

Runge–Kutta (RK) methods are another type of one-step discretization, a very popular choice

Aim to achieve higher order accuracy by combining evaluations of f (*i.e.* estimates of y') at several points in $[t_k, t_{k+1}]$

RK methods all fit within a general framework, which can be described in terms of Butcher tableaus

We will first consider two RK examples: two evaluations of f and four evaluations of f

Runge–Kutta Methods

The family of Runge–Kutta methods with two intermediate evaluations is defined by

$$y_{k+1} = y_k + h(ak_1 + bk_2),$$

where $k_1 = f(t_k, y_k)$, $k_2 = f(t_k + \alpha h, y_k + \beta h k_1)$

The Euler method is a member of this family, with $a = 1$ and $b = 0$
By careful analysis of the truncation error, it can be shown that we can choose a, b, α, β to obtain a second-order method

Runge–Kutta Methods

[*order2.py*] Three such examples are:

- The modified Euler method ($a = 0$, $b = 1$, $\alpha = \beta = 1/2$):

$$y_{k+1} = y_k + hf\left(t_k + \frac{1}{2}h, y_k + \frac{1}{2}hf(t_k, y_k)\right)$$

- The improved Euler method (or Heun's method) ($a = b = 1/2$, $\alpha = \beta = 1$):

$$y_{k+1} = y_k + \frac{1}{2}h[f(t_k, y_k) + f(t_k + h, y_k + hf(t_k, y_k))]$$

- Ralston's method ($a = 1/4$, $b = 3/4$, $\alpha = 2/3$, $\beta = 2/3$)

$$y_{k+1} = y_k + \frac{1}{4}h[f(t_k, y_k) + 3f(t_k + \tfrac{2h}{3}, y_k + \tfrac{2h}{3}f(t_k, y_k))]$$

Runge–Kutta Methods

The most famous Runge–Kutta method is the "classical fourth-order method", RK4 (used by MATLAB's `ode45`):

$$y_{k+1} = y_k + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

where

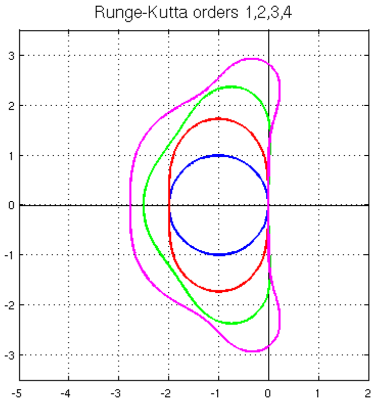
$$\begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + h/2, y_k + hk_1/2) \\ k_3 &= f(t_k + h/2, y_k + hk_2/2) \\ k_4 &= f(t_k + h, y_k + hk_3) \end{aligned}$$

Analysis of the truncation error in this case (which gets quite messy!) gives $T_k = O(h^4)$

Runge–Kutta Methods: Stability

We can also examine stability of RK4 methods for $y' = \lambda y$

Figure shows stability regions for four different RK methods (higher order RK methods have larger stability regions here)



Butcher tableau

Can summarize an $s+1$ stage Runge–Kutta method using a triangular grid of coefficients

$$\begin{array}{c|ccccc}
\alpha_0 & & & & & \\
\alpha_1 & \beta_{1,0} & & & & \\
\vdots & \vdots & & & & \\
\alpha_s & \beta_{s,0} & \beta_{s,1} & \dots & \beta_{s,s-1} & \\
\hline
 & \gamma_0 & \gamma_1 & \dots & \gamma_{s-1} & \gamma_s
\end{array}$$

The ith intermediate step is

$$f(t_k + \alpha_i h, y_k + h \sum_{j=0}^{i-1} \beta_{i,j} k_j).$$

The $(k+1)$th answer for y is

$$y_{k+1} = y_k + h \sum_{j=0}^{s} \gamma_j k_j.$$

Estimation of error

First approach: Richardson extrapolation
[*r_extrap.py*/*r_extrap2.py*]

Suppose that y_{k+2} is the numerical result of two steps with size h of a Runge–Kutta method of order p, and w is the result of one big step with step size $2h$. Then the error of y_{k+2} can be approximated as

$$y(t_k + 2h) - y_{k+2} = \frac{y_{k+2} - w}{2^p - 1} + O(h^{p+2})$$

and

$$\hat{y}_{k+2} = y_{k+2} + \frac{y_{k+2} - w}{2^p - 1}$$

is an approximation of order $p + 1$ to $y(t_0 + 2h)$.

Estimation of error

Second approach: can derive Butcher tableaus that contain an additional higher-order formula for estimating error. *e.g.* Fehlberg's order 4(5) method, RKF45

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{4} & \frac{1}{4} & & & & & \\
\frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & \\
\frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & \\
1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & & \\
\frac{1}{2} & \frac{-8}{27} & 2 & \frac{-3544}{2565} & \frac{1859}{4104} & \frac{-11}{40} & \\
\hline
y_{k+1} & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & 0 \\
\hat{y}_{k+1} & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55}
\end{array}
$$

y_{k+1} is order 4 and $\hat{y}_{k+1}$ is order 5. Use $y_{k+1} - \hat{y}_{k+1}$ as an error estimate.

Higher-order methods

Fehlberg's 7(8) method[7]

$$
\begin{array}{c|ccccccccccccc}
0 & & & & & & & & & & & & & \\
\frac{2}{27} & \frac{2}{27} & & & & & & & & & & & & \\
\frac{1}{9} & \frac{1}{36} & \frac{1}{12} & & & & & & & & & & & \\
\frac{1}{6} & \frac{1}{24} & 0 & \frac{1}{8} & & & & & & & & & & \\
\frac{5}{12} & \frac{5}{12} & 0 & -\frac{25}{16} & \frac{25}{16} & & & & & & & & & \\
\frac{1}{2} & \frac{1}{20} & 0 & 0 & \frac{1}{4} & \frac{1}{5} & & & & & & & & \\
\frac{5}{6} & -\frac{25}{108} & 0 & 0 & \frac{125}{108} & -\frac{65}{27} & \frac{125}{54} & & & & & & & \\
\frac{1}{6} & \frac{31}{300} & 0 & 0 & 0 & \frac{61}{225} & -\frac{2}{9} & \frac{13}{900} & & & & & & \\
\frac{2}{3} & 2 & 0 & 0 & -\frac{53}{6} & \frac{704}{45} & -\frac{107}{9} & \frac{67}{90} & 3 & & & & & \\
\frac{1}{3} & -\frac{91}{108} & 0 & 0 & \frac{23}{108} & -\frac{976}{135} & \frac{311}{54} & -\frac{19}{60} & \frac{17}{6} & -\frac{1}{12} & & & & \\
1 & \frac{2383}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{301}{82} & \frac{2133}{4100} & \frac{45}{82} & \frac{45}{164} & \frac{18}{41} & & & \\
0 & \frac{3}{205} & 0 & 0 & 0 & 0 & -\frac{6}{41} & -\frac{3}{205} & -\frac{3}{41} & \frac{3}{41} & \frac{6}{41} & 0 & & \\
1 & -\frac{1777}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{289}{82} & \frac{2193}{4100} & \frac{51}{82} & \frac{33}{164} & \frac{12}{41} & 0 & 1 & \\
\hline
y_{k+1} & \frac{41}{840} & 0 & 0 & 0 & 0 & \frac{34}{105} & \frac{9}{35} & \frac{9}{35} & \frac{9}{280} & \frac{9}{280} & \frac{41}{840} & 0 & 0 \\
\hat{y}_{k+1} & 0 & 0 & 0 & 0 & 0 & \frac{34}{105} & \frac{9}{35} & \frac{9}{35} & \frac{9}{280} & \frac{9}{280} & & \frac{41}{840} & \frac{41}{840}
\end{array}
$$

[7]From *Solving Ordinary Differential Equations* by Hairer, Nørsett, and Wanner.

Stiff systems

You may have heard of "stiffness" in the context of ODEs: an important, though somewhat fuzzy, concept

Common definition of stiffness for a linear ODE system $y' = Ay$ is that A has eigenvalues that differ greatly in magnitude[8]

The eigenvalues determine the time scales, and hence large differences in λ's $\implies$ resolve disparate timescales simultaneously!

[8]Nonlinear case: stiff if the Jacobian, J_f, has large differences in eigenvalues, but this defn. isn't always helpful since J_f changes at each time-step

Stiff systems

Suppose we're primarily interested in the long timescale. Then:

- We'd like to take large time steps and resolve the long timescale accurately
- But we may be forced to take extremely small timesteps to avoid instabilities due to the fast timescale

In this context it can be highly beneficial to use an implicit method since that enforces stability regardless of timestep size

Stiff systems

From a practical point of view, an ODE is stiff if there is a significant benefit in using an implicit instead of explicit method

e.g. this occurs if the time-step size required for stability is much smaller than size required for the accuracy level we want

Example [*stiff.py*/*stiff2.py*] : Consider $y' = Ay$, $y_0 = [1, 0]^T$ where

$$A = \begin{bmatrix} 998 & 1998 \\ -999 & -1999 \end{bmatrix}$$

which has $\lambda_1 = -1$, $\lambda_2 = -1000$ and exact solution

$$y(t) = \begin{bmatrix} 2e^{-t} - e^{-1000t} \\ -e^{-t} + e^{-1000t} \end{bmatrix}$$

Multistep Methods

So far we have looked at one-step methods, but to improve efficiency why not try to reuse data from earlier time-steps?

This is exactly what multistep methods do:

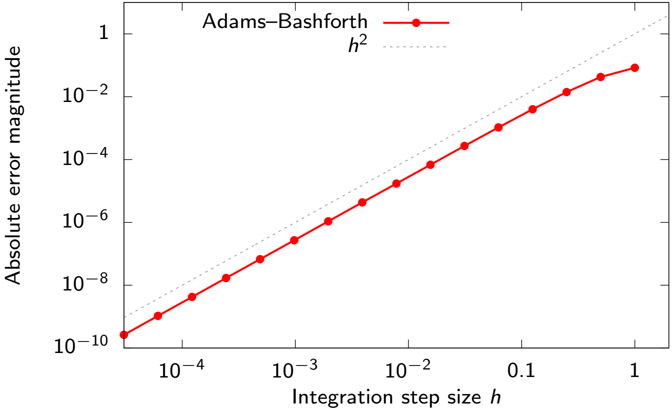
$$y_{k+1} = \sum_{i=1}^{m} \alpha_i y_{k+1-i} + h \sum_{i=0}^{m} \beta_i f(t_{k+1-i}, y_{k+1-i})$$

If $\beta_0 = 0$ then the method is explicit

We can derive the parameters by interpolating and then integrating the interpolant

Multistep Methods

Python example: [*ad-bash.py*] Second-order Adams–Bashforth scheme



Multistep Methods

The stability of multistep methods, often called “zero stability,” is an interesting topic, but not considered here

Question: Multistep methods require data from several earlier time-steps, so how do we initialize?

Answer: The standard approach is to start with a one-step method and move to multistep once there is enough data

Some key advantages of one-step methods:

- They are “self-starting”
- Easier to adapt time-step size

ODE Boundary Value Problems

Consider the ODE Boundary Value Problem (BVP):[9] find $u \in C^2[a, b]$ such that

$$-\alpha u''(x) + \beta u'(x) + \gamma u(x) = f(x), \quad x \in [a, b]$$

for $\alpha, \beta, \gamma \in \mathbb{R}$ and $f : \mathbb{R} \to \mathbb{R}$

The terms in this ODE have standard names:

$-\alpha u''(x)$:	diffusion term
$\beta u'(x)$:	convection (or transport) term
$\gamma u(x)$:	reaction term
$f(x)$:	source term

[9]Often called a "Two-point boundary value problem"

ODE BVPs

Also, since this is a BVP u must satisfy some boundary conditions, *e.g.* $u(a) = c_1$, $u(b) = c_2$

$u(a) = c_1$, $u(b) = c_2$ are called Dirichlet boundary conditions

Can also have:

- A Neumann boundary condition: $u'(b) = c_2$
- A Robin (or "mixed") boundary condition:[10] $u'(b) + c_2 u(b) = c_3$

[10]With $c_2 = 0$, this is a Neumann condition

ODE BVPs

This is an ODE, so we could try to use the ODE IVP solvers to solve it!

Question: How would we make sure the solution satisfies $u(b) = c_2$?

ODE BVPs

Answer: Solve the IVP with $u(a) = c_1$ and $u'(a) = s_0$, and then update s_k iteratively for $k = 1, 2, \ldots$ until $u(b) = c_2$ is satisfied

This is called the “shooting method”, we picture it as shooting a projectile to hit a target at $x = b$

However, the shooting method does not generalize to PDEs hence it is not broadly useful

ODE BVPs

A more general approach is to formulate a coupled system of equations for the BVP based on a finite difference approximation

Suppose we have a grid $x_i = a + ih$, $i = 0, 1, \ldots, n-1$, where $h = (b-a)/(n-1)$

Then our approximation to $u \in C^2[a, b]$ is represented by a vector $U \in \mathbb{R}^n$, where $U_i \approx u(x_i)$

ODE BVPs

Recall the ODE:

$$-\alpha u''(x) + \beta u'(x) + \gamma u(x) = f(x), \quad x \in [a, b]$$

Let's develop an approximation for each term in the ODE

For the reaction term γu, we have the pointwise approximation
$\gamma U_i \approx \gamma u(x_i)$

ODE BVPs

Similarly, for the derivative terms:

- Let $D_2 \in \mathbb{R}^{n\times n}$ denote diff. matrix for the second derivative
- Let $D_1 \in \mathbb{R}^{n\times n}$ denote diff. matrix for the first derivative

Then $-\alpha(D_2 U)_i \approx -\alpha u''(x_i)$ and $\beta(D_1 U)_i \approx \beta u'(x_i)$

Hence, we obtain $(AU)_i \approx -\alpha u''(x_i) + \beta u'(x_i) + \gamma u(x_i)$, where $A \in \mathbb{R}^{n\times n}$ is

$$A \equiv -\alpha D_2 + \beta D_1 + \gamma \mathrm{I}$$

Similarly, we represent the right hand side by sampling f at the grid points, hence we introduce $F \in \mathbb{R}^n$, where $F_i = f(x_i)$

ODE BVPs

Therefore, we obtain the linear[11] system for $U \in \mathbb{R}^n$:

$$AU = F$$

Hence, we have converted a linear differential equation into a linear algebraic equation

(Similarly we can convert a nonlinear differential equation into a nonlinear algebraic system)

However, we are not finished yet, need to account for the boundary conditions!

[11] It is linear here since the ODE BVP is linear

ODE BVPs

Dirichlet boundary conditions: we need to impose $U_0 = c_1$, $U_{n-1} = c_2$

Since we fix U_0 and U_{n-1}, they are no longer variables: we should eliminate them from our linear system

However, instead of removing rows and columns from A, it is slightly simpler from the implementational point of view to:

- "zero out" first row of A, then set $A(0,0) = 1$ and $F_0 = c_1$
- "zero out" last row of A, then set $A(n-1, n-1) = 1$ and $F_{n-1} = c_2$

ODE BVPs

We can implement the above strategy for $AU = F$ in Python

Useful trick[12] for checking your code:

1. choose a solution u that satisfies the BCs
2. substitute into the ODE to get a right-hand side f
3. compute the ODE approximation with f from step 2
4. verify that you get the expected convergence rate for the approximation to u

e.g. consider $x \in [0,1]$ and set $u(x) = e^x \sin(2\pi x)$:

$$\begin{aligned} f(x) &\equiv -\alpha u''(x) + \beta u'(x) + \gamma u(x) \\ &= -\alpha e^x \left[4\pi \cos(2\pi x) + (1-4\pi^2)\sin(2\pi x)\right] + \\ &\quad \beta e^x \left[\sin(2\pi x) + 2\pi \cos(2\pi x)\right] + \gamma e^x \sin(2\pi x) \end{aligned}$$

[12]Sometimes called the "method of manufactured solutions"

ODE BVPs

Python example: [*ode_bvp.py*] ODE BVP via finite differences

Convergence results:

h	error
2.0×10^{-2}	5.07×10^{-3}
1.0×10^{-2}	1.26×10^{-3}
5.0×10^{-3}	3.17×10^{-4}
2.5×10^{-3}	7.92×10^{-5}

$O(h^2)$, as expected due to second order differentiation matrices

ODE BVPs: BCs involving derivatives

Question: How would we impose the Robin boundary condition $u'(b) + c_2 u(b) = c_3$, and preserve the $O(h^2)$ convergence rate?

Option 1: Introduce a "ghost node" at $x_n = b + h$, this node is involved in both the B.C. and the $(n-1)^{\text{th}}$ matrix row

Employ central difference approx. to $u'(b)$ to get approx. B.C.:

$$\frac{U_n - U_{n-2}}{2h} + c_2 U_{n-1} = c_3,$$

or equivalently

$$U_n = U_{n-2} - 2hc_2 U_{n-1} + 2hc_3$$

ODE BVPs: BCs involving derivatives

The $(n-1)^{\text{th}}$ equation is

$$-\alpha\frac{U_{n-2}-2U_{n-1}+U_n}{h^2}+\beta\frac{U_n-U_{n-2}}{2h}+\gamma U_{n-1}=F_{n-1}$$

We can substitute our expression for U_n into the above equation, and hence eliminate U_n:

$$\left(-\frac{2\alpha c_3}{h}+\beta c_3\right)-\frac{2\alpha}{h^2}U_{n-2}+\left(\frac{2\alpha}{h^2}(1+hc_2)-\beta c_2+\gamma\right)U_{n-1}=F_{n-1}$$

Set $F_{n-1}\leftarrow F_{n-1}-\left(-\frac{2\alpha c_3}{h}+\beta c_3\right)$, we get $n\times n$ system $AU=F$

Option 2: Use a one-sided finite-difference formula for $u'(b)$ in the Robin BC

PDEs

As discussed in the introduction, it is a natural extension to consider Partial Differential Equations (PDEs)

There are three main classes of PDEs:[13]

equation type	prototypical example	equation
hyperbolic	wave equation	$u_{tt} - u_{xx} = 0$
parabolic	heat equation	$u_t - u_{xx} = f$
elliptic	Poisson equation	$u_{xx} + u_{yy} = f$

Question: Where do these names come from?

[13]Notation: $u_x \equiv \frac{\partial u}{\partial x}$, $u_{xy} \equiv \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x}\right)$

PDEs

Answer: The names are related to conic sections

General second-order PDEs have the form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0$$

This “looks like” the quadratic function

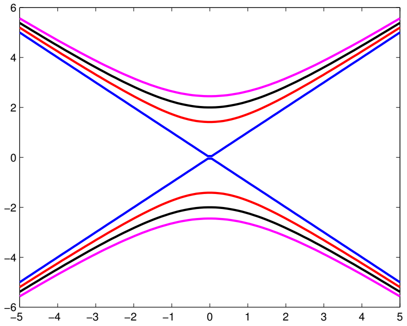
$$q(x, y) = ax^2 + bxy + cy^2 + dx + ey$$

PDEs: Hyperbolic

Wave equation: $u_{tt} - u_{xx} = 0$

Corresponding quadratic function is $q(x, t) = t^2 - x^2$

$q(x, t) = c$ gives a hyperbola, *e.g.* for $c = 0 : 2 : 6$, we have

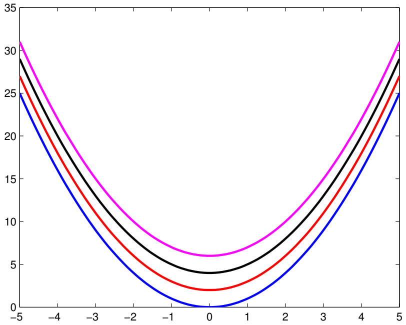


PDEs: Parabolic

Heat equation: $u_t - u_{xx} = 0$

Corresponding quadratic function is $q(x,t) = t - x^2$

$q(x,t) = c$ gives a parabola, *e.g.* for $c = 0 : 2 : 6$, we have

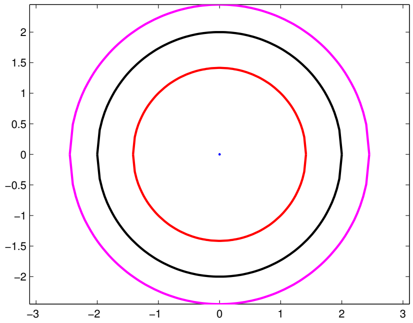


PDEs: Elliptic

Poisson equation: $u_{xx} + u_{yy} = f$

Corresponding quadratic function is $q(x, y) = x^2 + y^2$

$q(x, y) = c$ gives an ellipse, *e.g.* for $c = 0 : 2 : 6$, we have



PDEs

In general, it is not so easy to classify PDEs using conic section naming

Many problems don't strictly fit into the classification scheme (*e.g.* nonlinear, or higher order, or variable coefficient equations)

Nevertheless, the names hyperbolic, parabolic, elliptic are the standard ways of describing PDEs, based on the criteria:

- Hyperbolic: time-dependent, conservative physical process, no steady state
- Parabolic: time-dependent, dissipative physical process, evolves towards steady state
- Elliptic: describes systems at equilibrium/steady-state

Hyperbolic PDEs

We introduced the wave equation $u_{tt} - u_{xx} = 0$ above

Note that the system of first order PDEs

$$\begin{aligned} u_t + v_x &= 0 \\ v_t + u_x &= 0 \end{aligned}$$

is equivalent to the wave equation, since

$$u_{tt} = (u_t)_t = (-v_x)_t = -(v_t)_x = -(-u_x)_x = u_{xx}$$

(This assumes that u, v are smooth enough for us to switch the order of the partial derivatives)

Hyperbolic PDEs

Hence we shall focus on the so-called linear advection equation

$$u_t + cu_x = 0$$

with initial condition $u(x, 0) = u_0(x)$, and $c \in \mathbb{R}$

This equation is representative of hyperbolic PDEs in general

It's a first order PDE, hence doesn't fit our conic section description, but it is:

- time-dependent
- conservative
- not evolving toward steady state

$\implies$ hyperbolic!

Hyperbolic PDEs

We can see that $u(x,t) = u_0(x - ct)$ satisfies the PDE

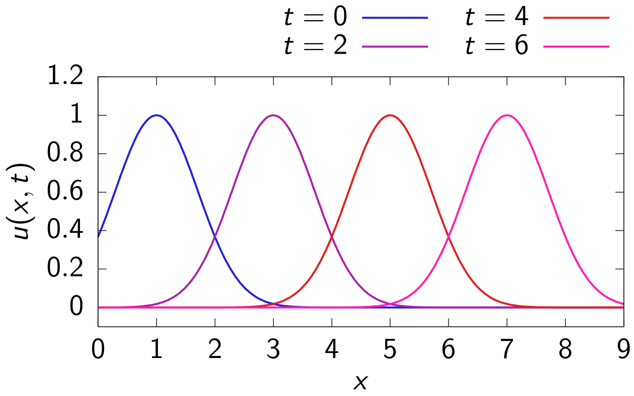
Let $z(x,t) \equiv x - ct$, then from the chain rule we have

$$\begin{aligned}
\frac{\partial}{\partial t} u_0(x-ct) + c\frac{\partial}{\partial x} u_0(x-ct) &= \frac{\partial}{\partial t} u_0(z(x,t)) + c\frac{\partial}{\partial x} u_0(z(x,t)) \\
&= u_0'(z)\frac{\partial z}{\partial t} + c u_0'(z)\frac{\partial z}{\partial x} \\
&= -c u_0'(z) + c u_0'(z) \\
&= 0
\end{aligned}$$

Hyperbolic PDEs

This tells us that the solution transports (or advects) the initial condition with "speed" c

e.g. with $c = 1$ and an initial condition $u_0(x) = e^{-(1-x)^2}$ we have:



Hyperbolic PDEs

We can understand the behavior of hyperbolic PDEs in more detail by considering characteristics

Characteristics are paths in the xt-plane — denoted by $(X(t), t)$ — on which the solution is constant

For $u_t + cu_x = 0$ we have $X(t) = X_0 + ct$,[14] since

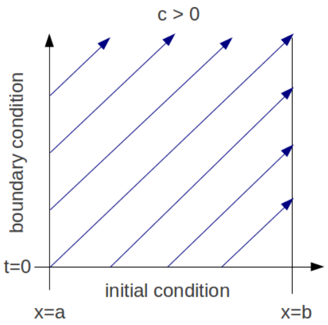
$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}u(X(t), t) &= u_t(X(t), t) + u_x(X(t), t)\frac{\mathrm{d}X(t)}{\mathrm{d}t} \\
&= u_t(X(t), t) + cu_x(X(t), t) \\
&= 0
\end{aligned}$$

[14]Each different choice of X_0 gives a distinct characteristic curve

Hyperbolic PDEs

Hence $u(X(t), t) = u(X(0), 0) = u_0(X_0)$, *i.e.* the initial condition is transported along characteristics

Characteristics have important implications for the direction of flow of information, and for boundary conditions

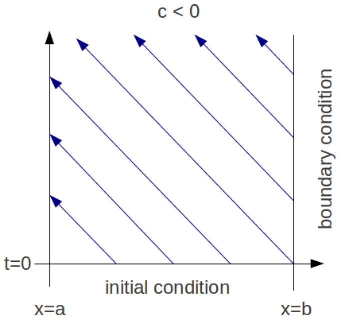


Must impose BC at $x = a$, cannot impose BC at $x = b$

Hyperbolic PDEs

Hence $u(X(t), t) = u(0, X(0)) = u_0(X_0)$, *i.e.* the initial condition is transported along characteristics

Characteristics have important implications for the direction of flow of information, and for boundary conditions



Must impose BC at $x = b$, cannot impose BC at $x = a$

Hyperbolic PDEs: More Complicated Characteristics

More generally, if we have a non-zero right-hand side in the PDE, then the situation is a bit more complicated on each characteristic

Consider $u_t + cu_x = f(t, x, u(t, x))$, and $X(t) = X_0 + ct$

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}u(X(t), t) &= u_t(X(t), t) + u_x(X(t), t)\frac{\mathrm{d}X(t)}{\mathrm{d}t} \\
&= u_t(X(t), t) + cu_x(X(t), t) \\
&= f(t, X(t), u(X(t), t))
\end{aligned}$$

In this case, the solution is no longer constant on $(X(t), t)$, but we have reduced a PDE to a set of ODEs, so that:

$$u(X(t), t) = u_0(X_0) + \int_0^t f(t, X(t), u(X(t), t)\mathrm{d}t$$

Hyperbolic PDEs: More Complicated Characteristics

We can also find characteristics for variable coefficient advection

Exercise: Verify that the characteristic curve for $u_t + c(t, x)u_x = 0$ is given by

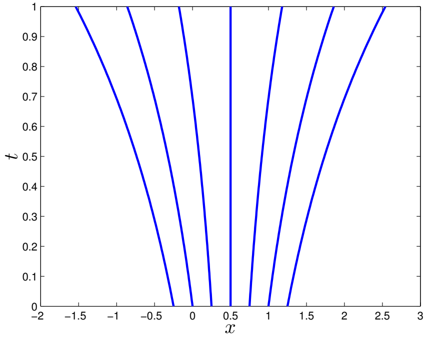
$$\frac{\mathrm{d}X(t)}{\mathrm{d}t} = c(X(t), t)$$

In this case, we have to solve an ODE to obtain the curve $(X(t), t)$ in the xt-plane

Hyperbolic PDEs: More Complicated Characteristics

e.g. for $c(t,x) = x - 1/2$, we get $X(t) = 1/2 + (X_0 - 1/2)e^t$

In this case, the characteristics "bend away" from $x = 1/2$



Characteristics also apply to nonlinear hyperbolic PDEs (*e.g.* Burger's equation), but this is outside the scope of AM205

Hyperbolic PDEs: Numerical Approximation

We now consider how to solve $u_t + cu_x = 0$ equation using a finite difference method

Question: Why finite differences? Why not just use characteristics?

Answer: Characteristics actually are a viable option for computational methods, and are used in practice

However, characteristic methods can become very complicated in 2D or 3D, or for nonlinear problems

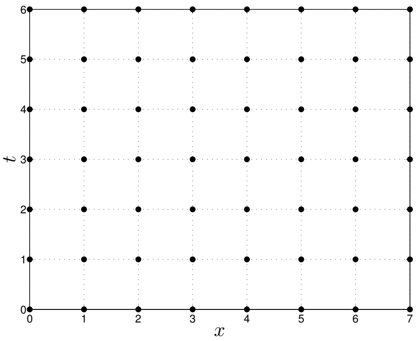
Finite differences are a much more practical choice in most circumstances

Hyperbolic PDEs: Numerical Approximation

Advection equation is an Initial Boundary Value Problem (IBVP)

We impose an initial condition, and a boundary condition (only one BC since first order PDE)

A finite difference approximation leads to a grid in the *xt*-plane



Hyperbolic PDEs: Numerical Approximation

The first step in developing a finite difference approximation for the advection equation is to consider the CFL condition[15]

The CFL condition is a necessary condition for the convergence of a finite difference approximation of a hyperbolic problem

Suppose we discretize $u_t + cu_x = 0$ in space and time using the explicit (in time) scheme

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\frac{U_j^n - U_{j-1}^n}{\Delta x} = 0$$

Here $U_j^n \approx u(t_n, x_j)$, where $t_n = n\Delta t$, $x_j = j\Delta x$

[15]Courant–Friedrichs–Lewy condition, published in 1928

Hyperbolic PDEs: Numerical Approximation

This can be rewritten as

$$\begin{aligned} U_j^{n+1} &= U_j^n - \frac{c\Delta t}{\Delta x}(U_j^n - U_{j-1}^n) \\ &= (1-\nu)U_j^n + \nu U_{j-1}^n \end{aligned}$$

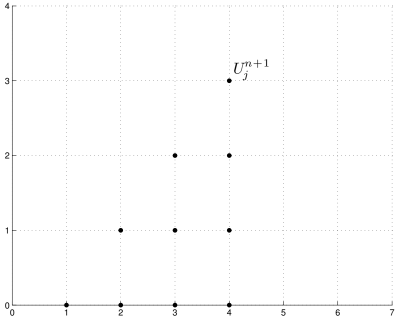
where

$$\nu \equiv \frac{c\Delta t}{\Delta x}$$

We can see that U_j^{n+1} depends only on U_j^n and U_{j-1}^n

Hyperbolic PDEs: Numerical Approximation

Definition: Domain of dependence of U_j^{n+1} is the set of values that U_j^{n+1} depends on



Hyperbolic PDEs: Numerical Approximation

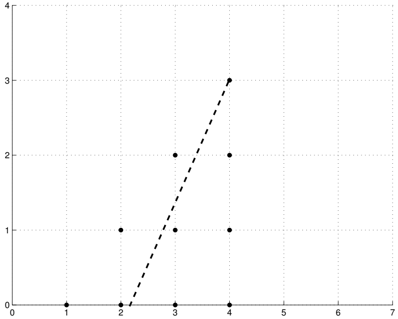
The domain of dependence of the exact solution $u(t_{n+1}, x_j)$ is determined by the characteristic curve passing through (t_{n+1}, x_j)

CFL Condition:

> For a convergent scheme, the domain of dependence of the PDE must lie within the domain of dependence of the numerical method

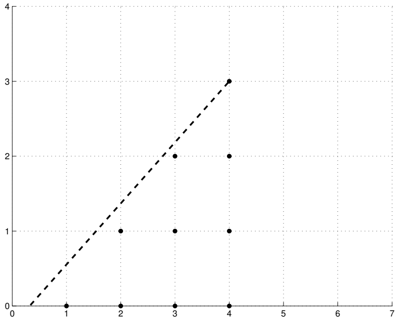
Hyperbolic PDEs: Numerical Approximation

Suppose the dashed line indicates characteristic passing through (t_{n+1}, x_j), then the scheme below satisfies the CFL condition



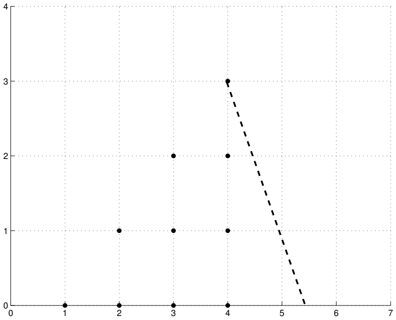
Hyperbolic PDEs: Numerical Approximation

The scheme below does not satisfy the CFL condition



Hyperbolic PDEs: Numerical Approximation

The scheme below does not satisfy the CFL condition (here $c < 0$)



Hyperbolic PDEs: Numerical Approximation

Question: What goes wrong if the CFL condition is violated?

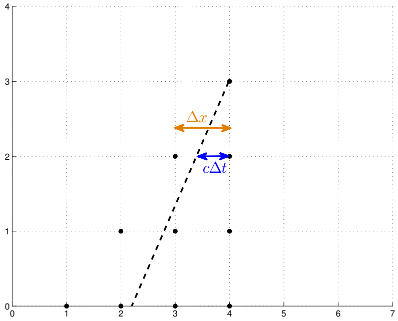
Hyperbolic PDEs: Numerical Approximation

Answer: The exact solution $u(x, t)$ depends on initial value $u_0(x_0)$, which is outside the numerical method's domain of dependence

Therefore, the numerical approx. to $u(x, t)$ is "insensitive" to the value $u_0(x_0)$, which means that the method cannot be convergent

Hyperbolic PDEs: Numerical Approximation

If $c > 0$, then we require $\nu \equiv \frac{c\Delta t}{\Delta x} \leq 1$ in $(*)$ for CFL to be satisfied



Hyperbolic PDEs: Numerical Approximation

Note that CFL is only a necessary condition for convergence

Its great value is its simplicity: CFL allows us to easily reject F.D. schemes for hyperbolic problems with very little investigation

For example, for $u_t + cu_x = 0$, the scheme

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\frac{U_j^n - U_{j-1}^n}{\Delta x} = 0 \qquad (*)$$

cannot be convergent if $c < 0$

Question: What small change to $(*)$ would give a better method when $c < 0$?

Hyperbolic PDEs: Upwind method

As foreshadowed earlier, we should pick our method to reflect the direction of propagation of information

This motivates the upwind scheme for $u_t + cu_x = 0$

$$U_j^{n+1} = \begin{cases} U_j^n - c\frac{\Delta t}{\Delta x}(U_j^n - U_{j-1}^n), & \text{if } c > 0 \\ U_j^n - c\frac{\Delta t}{\Delta x}(U_{j+1}^n - U_j^n), & \text{if } c < 0 \end{cases}$$

The upwind scheme satisfies CFL condition if $|\nu| \equiv |c\Delta t/\Delta x| \leq 1$

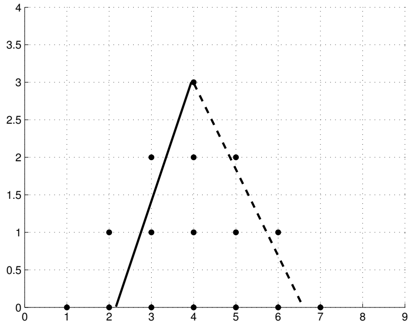
ν is often called the CFL number

Hyperbolic PDEs: Central difference method

Another method that seems appealing is the central difference method:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x} = 0$$

This satisfies CFL for $|\nu| \equiv |c\Delta t/\Delta x| \leq 1$, regardless of sign$(c)$



We shall see shortly, however, that this is a bad method!

Hyperbolic PDEs: Accuracy

Recall that truncation error is "what is left over when we substitute exact solution into the numerical approximation"

Truncation error is analogous for PDEs, *e.g.* for the $(c > 0)$ upwind method, truncation error is:

$$T_j^n \equiv \frac{u(t^{n+1}, x_j) - u(t^n, x_j)}{\Delta t} + c\frac{u(t^n, x_j) - u(t^n, x_{j-1})}{\Delta x}$$

The order of accuracy is then the largest p such that

$$T_j^n = O((\Delta x)^p + (\Delta t)^p)$$

Hyperbolic PDEs: Accuracy

See Lecture: For the upwind method, we have

$$T_j^n = \frac{1}{2}\left[\Delta t\, u_{tt}(t^n, x_j) - c\Delta x\, u_{xx}(t^n, x_j)\right] + \text{H.O.T.}$$

Hence the upwind scheme is first order accurate

Hyperbolic PDEs: Accuracy

Just like with ODEs, truncation error is related to convergence in the limit $\Delta t, \Delta x \to 0$

Note that to let $\Delta t, \Delta x \to 0$, we generally need to decide on a relationship between Δt and Δx

e.g. to let $\Delta t, \Delta x \to 0$ for the upwind scheme, we would set $\frac{c\Delta t}{\Delta x} = \nu \in (0, 1]$; this ensures CFL is satisfied for all $\Delta x, \Delta t$

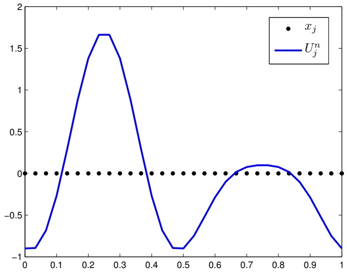
Hyperbolic PDEs: Accuracy

In general, convergence of a finite difference method for a PDE is related to both its truncation error and its stability

We'll discuss this in more detail shortly, but first we consider how to analyze stability via Fourier stability analysis

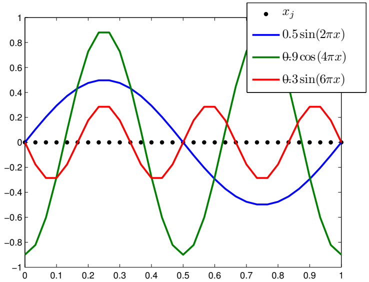
Hyperbolic PDEs: Stability

Let's suppose that U_j^n is periodic on the grid $x_1, x_2, \ldots, x_n$



Hyperbolic PDEs: Stability

Then we can represent U_j^n as a linear combination of sin and cos functions, *i.e.* Fourier modes



Or, equivalently, as a linear combination of complex exponentials, since $e^{ikx} = \cos(kx) + i\sin(kx)$ so that

$$\sin(x) = \frac{1}{2i}(e^{ix} - e^{-ix}), \qquad \cos(x) = \frac{1}{2}(e^{ix} + e^{-ix})$$

Hyperbolic PDEs: Stability

For simplicity, let's just focus on only one of the Fourier modes

In particular, we consider the ansatz $U_j^n(k) \equiv \lambda(k)^n e^{ikx_j}$, where k is the wave number and $\lambda(k) \in \mathbb{C}$

Key idea: Suppose that $U_j^n(k)$ satisfies our finite difference equation, then this will allow us to solve[16] for $\lambda(k)$

The value of $|\lambda(k)|$ indicates whether the Fourier mode e^{ikx_j} is amplified or damped

If $|\lambda(k)| \leq 1$ for all k then the scheme does not amplify any Fourier modes $\implies$ stable!

[16] In general a solution for $\lambda(k)$ exists, which justifies our choice of ansatz

Hyperbolic PDEs: Stability

We now perform Fourier stability analysis for the $(c > 0)$ upwind scheme (recall that $\nu = \frac{c\Delta t}{\Delta x}$):

$$U_j^{n+1} = U_j^n - \nu(U_j^n - U_{j-1}^n)$$

Substituting in $U_j^n(k) = \lambda(k)^n e^{ik(j\Delta x)}$ gives

$$\begin{aligned}\lambda(k)e^{ik(j\Delta x)} &= e^{ik(j\Delta x)} - \nu(e^{ik(j\Delta x)} - e^{ik((j-1)\Delta x)}) \\ &= e^{ik(j\Delta x)} - \nu e^{ik(j\Delta x)}(1 - e^{-ik\Delta x})\end{aligned}$$

Hence

$$\lambda(k) = 1 - \nu(1 - e^{-ik\Delta x}) = 1 - \nu(1 - \cos(k\Delta x) + i\sin(k\Delta x))$$

Hyperbolic PDEs: Stability

It follows that

$$\begin{aligned} |\lambda(k)|^2 &= [(1-\nu) + \nu\cos(k\Delta x)]^2 + [\nu\sin(k\Delta x)]^2 \\ &= (1-\nu)^2 + \nu^2 + 2\nu(1-\nu)\cos(k\Delta x) \\ &= 1 - 2\nu(1-\nu)(1-\cos(k\Delta x)) \end{aligned}$$

and from the trig. identity $(1-\cos(\theta)) = 2\sin^2(\frac{\theta}{2})$, we have

$$|\lambda(k)|^2 = 1 - 4\nu(1-\nu)\sin^2\left(\frac{1}{2}k\Delta x\right)$$

Due to the CFL condition, we first suppose that $0 \le \nu \le 1$

It then follows that $0 \le 4\nu(1-\nu)\sin^2\left(\frac{1}{2}k\Delta x\right) \le 1$, and hence $|\lambda(k)| \le 1$

Hyperbolic PDEs: Stability

In contrast, consider stability of the central difference approx.:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} + c\frac{U_{j+1}^n - U_{j-1}^n}{2\Delta x} = 0$$

Recall that this also satisfies the CFL condition as long as $|\nu| \leq 1$

But Fourier stability analysis yields

$$\lambda(k) = 1 - \nu i \sin(k\Delta x) \implies |\lambda(k)|^2 = 1 + \nu^2 \sin^2(k\Delta x)$$

and hence $|\lambda(k)| > 1$ (unless $\sin(k\Delta x) = 0$), *i.e.* unstable!

Consistency

We say that a numerical scheme is consistent with a PDE if its truncation error tends to zero as $\Delta x, \Delta t \to 0$

For example, any first (or higher) order scheme is consistent

Lax Equivalence Theorem

Then a fundamental theorem in Scientific Computing is the Lax[17] Equivalence Theorem:

$$\boxed{\text{For a consistent finite difference approx. to a linear evolutionary problem, the stability of the scheme is necessary and sufficient for convergence}}$$

This theorem refers to linear evolutionary problems, *e.g.* linear hyperbolic or parabolic PDEs

[17] Peter Lax, Courant Institute, NYU

Lax Equivalence Theorem

We know how to check consistency: Derive the truncation error

We know how to check stability: Fourier stability analysis

Hence, from Lax, we have a general approach for verifying convergence

Also, as with ODEs, convergence rate is determined by truncation error

Lax Equivalence Theorem

Note that strictly speaking Fourier stability analysis only applies for periodic problems

However, it can be shown that conclusions of Fourier stability analysis hold true more generally

Hence Fourier stability analysis is the standard tool for examining stability of finite-difference methods for PDEs

Python example: [*transp.py*/*transp2.py*] One-sided and centered-difference discretizations of the transport equation

Hyperbolic PDEs: Semi-discretization

So far, we have developed full discretizations (both space and time) of the advection equation, and considered accuracy and stability

However, it can be helpful to consider semi-discretizations, where we discretize only in space, or only in time

For example, discretizing $u_t + c(t,x)u_x = 0$ in space[18] using a backward difference formula gives

$$\frac{\partial U_j(t)}{\partial t} + c_j(t)\frac{U_j(t) - U_{j-1}(t)}{\Delta x} = 0, \qquad j = 1, \ldots, n$$

[18] Here we show an example where c is not constant

Hyperbolic PDEs: Semi-discretization

This gives a system of ODEs, $U_t = f(t, U(t))$, where $U(t) \in \mathbb{R}^n$ and

$$f_j(t, U(t)) \equiv -c_j(t)\frac{U_j(t) - U_{j-1}(t)}{\Delta x}$$

We could approximate this ODE using forward Euler (to get our upwind scheme):

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = f(t^n, U^n) = -c_j^n \frac{U_j^n - U_{j-1}^n}{\Delta x}$$

Or backward Euler:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = f(t^{n+1}, U^{n+1}) = -c_j^{n+1} \frac{U_j^{n+1} - U_{j-1}^{n+1}}{\Delta x}$$

Hyperbolic PDEs: Method of Lines

Or we could use a "black box" ODE solver, such as `ode45`, to solve the system of ODEs

This "black box" approach is called the method of lines [*m_of_lines.py*]

The name "lines" is because we solve each $U_j(t)$ for a fixed x_j, *i.e.* a line in the xt-plane

With method of lines we let the ODE solver to choose step sizes Δt to obtain a stable and accurate scheme

The Wave Equation

We now briefly return to the wave equation:

$$u_{tt} - c^2 u_{xx} = 0$$

In one spatial dimension, this models, say, vibrations in a taut string

The Wave Equation

Many schemes have been proposed for the wave equation

One good option is to use central difference approximations[19] for both u_{tt} and u_{xx}:

$$\frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{\Delta t^2} - c^2 \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{\Delta x^2} = 0$$

Key points:

- Truncation error analysis $\implies$ second-order accurate
- Fourier stability analysis $\implies$ stable for $0 \leq c\Delta t/\Delta x \leq 1$
- Two-step method in time, need a one-step method to "get started"

[19] Can arrive at the same result by discretizing the equivalent first order system

The Heat Equation

The canonical parabolic equation is the heat equation

$$u_t - \alpha u_{xx} = f(t, x),$$

where α models thermal diffusivity

In this section, we shall omit α for convenience

Note that this is an Initial-Boundary Value Problem:

- We impose an initial condition $u(0, x) = u_0(x)$
- We impose boundary conditions on both sides of the domain

The Heat Equation

A natural idea would be to discretize u_{xx} with a central difference, and employ the Euler method in time:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} - \frac{U_{j-1}^n - 2U_j^n + U_{j+1}^n}{\Delta x^2} = 0$$

Or we could use backward Euler in time:

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} - \frac{U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{\Delta x^2} = 0$$

The Heat Equation

Or we could do something "halfway in between":

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} - \frac{1}{2}\frac{U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{\Delta x^2} - \frac{1}{2}\frac{U_{j-1}^{n} - 2U_j^{n} + U_{j+1}^{n}}{\Delta x^2} = 0$$

This is called the Crank–Nicolson method[20]

In fact, it is common to consider a 1-parameter "family" of methods that include all of the above: the θ-method

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} - \theta\frac{U_{j-1}^{n+1} - 2U_j^{n+1} + U_{j+1}^{n+1}}{\Delta x^2} - (1-\theta)\frac{U_{j-1}^{n} - 2U_j^{n} + U_{j+1}^{n}}{\Delta x^2} = 0$$

where $\theta \in [0, 1]$

[20] From a paper by Crank and Nicolson in 1947, note: "Nicolson" is not a typo!

The Heat Equation

With the θ-method:

- $\theta = 0 \implies$ Euler
- $\theta = \frac{1}{2} \implies$ Crank–Nicolson
- $\theta = 1 \implies$ backward Euler

For the θ-method, we can

1. perform Fourier stability analysis
2. calculate the truncation error

The θ-Method: Stability

Fourier stability analysis: Set $U_j^n(k) = \lambda(k)^n e^{ik(j\Delta x)}$ to get

$$\lambda(k) = \frac{1 - 4(1-\theta)\mu \sin^2\left(\frac{1}{2}k\Delta x\right)}{1 + 4\theta\mu \sin^2\left(\frac{1}{2}k\Delta x\right)}$$

where $\mu \equiv \Delta t/(\Delta x)^2$

Here we cannot get $\lambda(k) > 1$, hence only concern is $\lambda(k) < -1$

Let's find conditions for stability, *i.e.* we want $\lambda(k) \geq -1$:

$$1 - 4(1-\theta)\mu \sin^2\left(\frac{1}{2}k\Delta x\right) \geq -\left[1 + 4\theta\mu \sin^2\left(\frac{1}{2}k\Delta x\right)\right]$$

The θ-Method: Stability

Or equivalently:

$$4\mu(1-2\theta)\sin^2\left(\frac{1}{2}k\Delta x\right) \le 2$$

For $\theta \in [0.5, 1]$ this inequality is always satisfied, hence the θ-method is unconditionally stable (*i.e.* stable independent of μ)

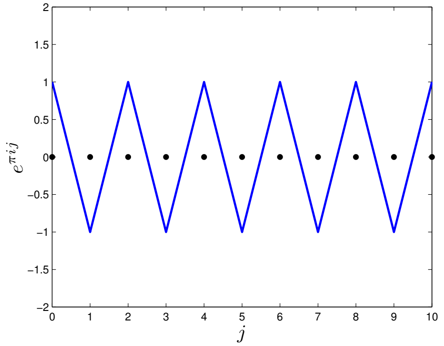
In the $\theta \in [0, 0.5)$ case, the "most unstable" Fourier mode is when $k = \pi/\Delta x$, since this maximizes the factor $\sin^2\left(\frac{1}{2}k\Delta x\right)$

The θ-Method: Stability

Note that this corresponds to the highest frequency mode that can be represented on our grid, since with $k = \pi/\Delta x$ we have

$$e^{ik(j\Delta x)} = e^{\pi ij} = (e^{\pi i})^j = (-1)^j$$

The $k = \pi/\Delta x$ mode:



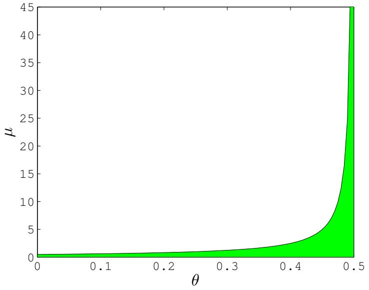
The θ-Method: Stability

This "sawtooth" mode is stable (and hence all modes are stable) if

$$4\mu(1-2\theta) \leq 2 \iff \mu \leq \frac{1}{2(1-2\theta)},$$

Hence for $\theta \in [0, 0.5)$, the θ-method is conditionally stable

The θ-Method: Stability



For $\theta \in [0, 0.5)$, θ-method is stable if μ is in the "green region,"
i.e. approaches unconditional stability as $\theta \to 0.5$

The θ-Method: Stability

Note that if we set θ to a value in $[0, 0.5)$, then stability time-step restriction is quite severe: $\Delta t \leq \frac{(\Delta x)^2}{2(1-2\theta)}$

Contrast this to the hyperbolic case where we had $\Delta t \leq \frac{\Delta x}{c}$

This is an indication that the system of ODEs that arise from spatially discretizing the heat equation are stiff

The θ-Method: Accuracy

The truncation error analysis is fairly involved, hence we just give the result:

$$\begin{aligned}
T_j^n &\equiv \frac{u_j^{n+1}-u_j^n}{\Delta t} - \theta\frac{u_{j-1}^{n+1}-2u_j^{n+1}+u_{j+1}^{n+1}}{\Delta x^2} - (1-\theta)\frac{u_{j-1}^{n}-2u_j^{n}+u_{j+1}^{n}}{\Delta x^2} \\
&= [u_t - u_{xx}] + \left[\left(\frac{1}{2}-\theta\right)\Delta t u_{xxt} - \frac{1}{12}(\Delta x)^2 u_{xxxx}\right] \\
&\quad + \left[\frac{1}{24}(\Delta t)^2 u_{ttt} - \frac{1}{8}(\Delta t)^2 u_{xxtt}\right] \\
&\quad + \left[\frac{1}{12}\left(\frac{1}{2}-\theta\right)\Delta t(\Delta x)^2 u_{xxxxt} - \frac{2}{6!}(\Delta x)^4 u_{xxxxxx}\right] + \cdots
\end{aligned}$$

The term $u_t - u_{xx}$ in T_j^n vanishes since u solves the PDE

The θ-Method: Accuracy

Key point: This is a first order method, unless $\theta = 1/2$, in which case we get a second order method!

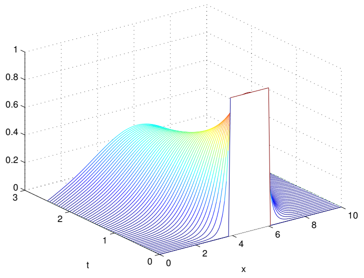
θ-method gives us consistency (at least first order) and stability (assuming Δt is chosen appropriately when $\theta \in [0, 1/2)$)

Hence, from Lax Equivalence Theorem, the method is convergent

The Heat Equation

Note that the heat equation models a diffusive process, hence it tends to smooth out discontinuities

Python demo: [*heat.py*/*c-n.py*] Heat equation with discontinous initial condition



This is very different to hyperbolic equations, *e.g.* the advection equation will just transport a discontinuity in u_0

Elliptic PDEs

Elliptic PDEs

The canonical elliptic PDE is the Poisson equation

In one-dimension, for $x \in [a, b]$, this is $-u''(x) = f(x)$ with boundary conditions at $x = a$ and $x = b$

We have seen this problem already: Two-point boundary value problem!

(Recall that Elliptic PDEs model steady-state behavior, there is no time-derivative)

Elliptic PDEs

In order to make this into a PDE, we need to consider more than one spatial dimension

Let $\Omega \subset \mathbb{R}^2$ denote our domain, then the Poisson equation for $(x, y) \in \Omega$ is

$$u_{xx} + u_{yy} = f(x, y)$$

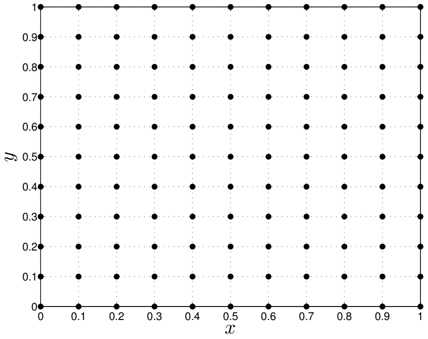
This is generally written more succinctly as $\Delta u = f$

We again need to impose boundary conditions (Dirichlet, Neumann, or Robin) on $\partial\Omega$ (recall $\partial\Omega$ denotes boundary of Ω)

Elliptic PDEs

We will consider how to use a finite difference scheme to approximate this 2D Poisson equation

First, we introduce a uniform grid to discretize Ω



Elliptic PDEs

Let $h = \Delta x = \Delta y$ denote the grid spacing

Then,

- $x_i = ih$, $i = 0, 1, 2 \ldots, n_x - 1$,
- $y_j = jh$, $j = 0, 1, 2, \ldots, n_y - 1$,
- $U_{i,j} \approx u(x_i, y_j)$

Then, we need to be able to approximate u_{xx} and u_{yy} on this grid

Natural idea: Use central difference approximation!

Elliptic PDEs

We have

$$u_{xx}(x_i, y_j) = \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} + O(h^2),$$

and

$$u_{yy}(x_i, y_j) = \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} + O(h^2),$$

so that

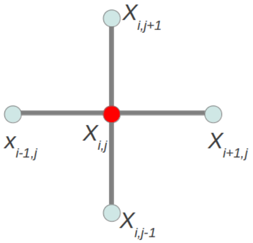
$$u_{xx}(x_i, y_j) + u_{yy}(x_i, y_j) = \frac{u(x_i, y_{j-1}) + u(x_{i-1}, y_j) - 4u(x_i, y_j) + u(x_{i+1}, y_j) + u(x_i, y_{j+1})}{h^2} + O(h^2)$$

Elliptic PDEs

Hence we define our approximation to the Laplacian as

$$\frac{U_{i,j-1} + U_{i-1,j} - 4U_{i,j} + U_{i+1,j} + U_{i,j+1}}{h^2}$$

This corresponds to a "5-point stencil"



Elliptic PDEs

As usual, we represent the numerical solution as a vector $\mathbb{U} \in \mathbb{R}^{n_x n_y}$

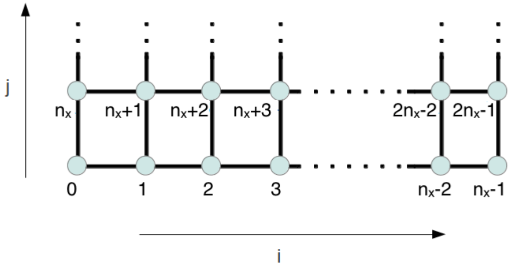
We want to construct a differentiation matrix $D_2 \in \mathbb{R}^{n_x n_y \times n_x n_y}$ that approximates the Laplacian

Question: How many non-zero diagonals will D_2 have?

To construct D_2, we need to be able to relate the entries of the vector $\mathbb{U}$ to the "2D grid-based values" $U_{i,j}$

Elliptic PDEs

Hence we need to number the nodes from 0 to $n_x n_y - 1$ — we number nodes along the "bottom row" first, then second bottom row, etc



Let $\mathcal{G}$ denote the mapping from the 2D indexing to the 1D indexing. From the above figure we have:

$$\mathcal{G}(i,j;n_x) = jn_x + i, \quad \text{and hence} \quad \mathbb{U}_{\mathcal{G}(i,j;n_x)} = U_{i,j}$$

Elliptic PDEs

Let us focus on node (i, j) in our F.D. grid, this corresponds to entry $\mathcal{G}(i, j; n_x)$ of $\mathbb{U}$

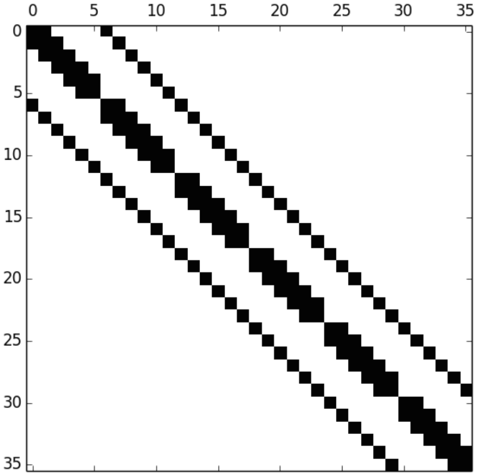
Due to the 5-point stencil, row $\mathcal{G}(i, j; n_x)$ of D_2 will only have non-zeros in columns

$$\begin{aligned}
\mathcal{G}(i, j-1; n_x) &= \mathcal{G}(i, j; n_x) - n_x, && (1)\\
\mathcal{G}(i-1, j; n_x) &= \mathcal{G}(i, j; n_x) - 1, && (2)\\
\mathcal{G}(i, j; n_x) &= \mathcal{G}(i, j; n_x), && (3)\\
\mathcal{G}(i+1, j; n_x) &= \mathcal{G}(i, j; n_x) + 1, && (4)\\
\mathcal{G}(i, j+1; n_x) &= \mathcal{G}(i, j; n_x) + n_x && (5)
\end{aligned}$$

- (2), (3), (4), give the same tridiagonal structure that we're used to from differentiation matrices in 1D domains
- (1), (5) give diagonals shifted by $\pm n_x$

Elliptic PDEs

For example, sparsity pattern of D_2 when $n_x = n_y = 6$

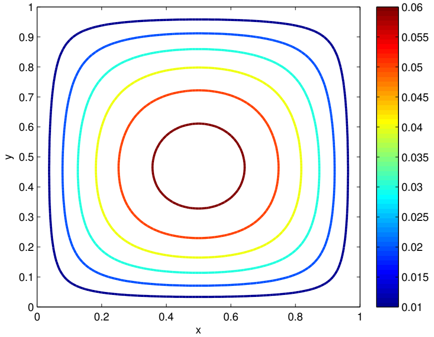


Elliptic PDEs

Python demo: [*poisson.py*] Solve the Poisson equation

$$\Delta u = -\exp\left\{-(x-0.25)^2 - (y-0.5)^2\right\},$$

for $(x, y) \in \Omega = [0, 1]^2$ with $u = 0$ on $\partial\Omega$



x
y
0
0.2
0.4
0.6
0.8
1
0.1
0.3
0.5
0.7
0.9
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045
0.05
0.055
0.06