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

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

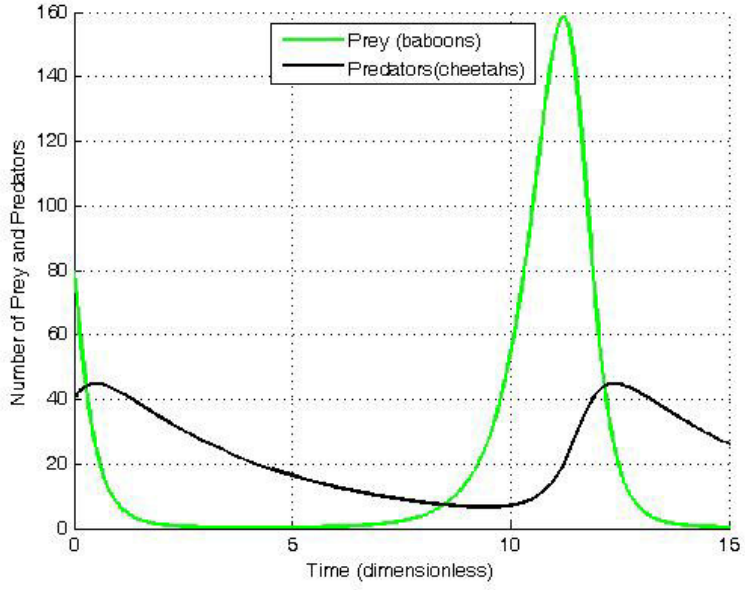
and $y(0) = y_0$, $v(0) = v_0$

ODE IVPs: A Predator–Prey ODE Model

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:[2]

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

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

[2] 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:

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

Again, this gives Euler's method:

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

Python example: 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[3]

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

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

[3] 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.[4]

[4] 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

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

Finally, we use the bound[5] $1 + hL_f \leq \exp(hL_f)$ to get the desired result. $\square$

[5] 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 $|f(x) - f(y)| = ||x| - |y|| \leq |x - y|$,[6] hence $L_f = 1$ in this case

[6] 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:

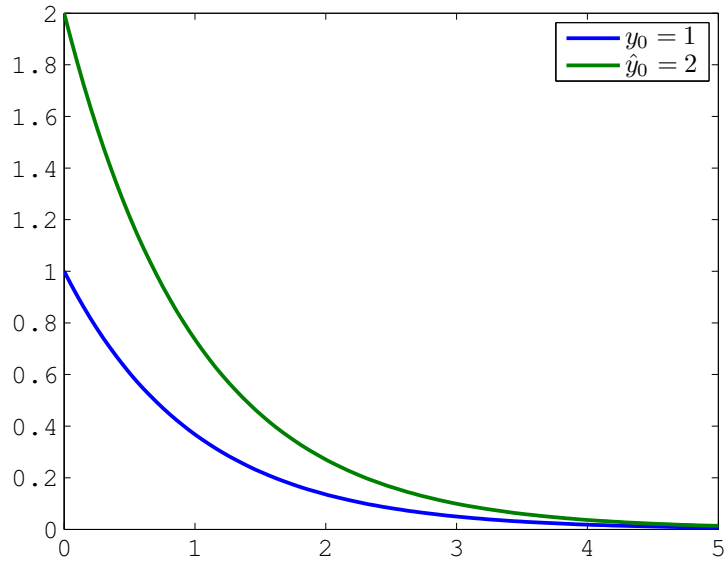
- We previously referred to this type of property as the conditioning of the problem
- Stability previously referred only to properties of a numerical approximation

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

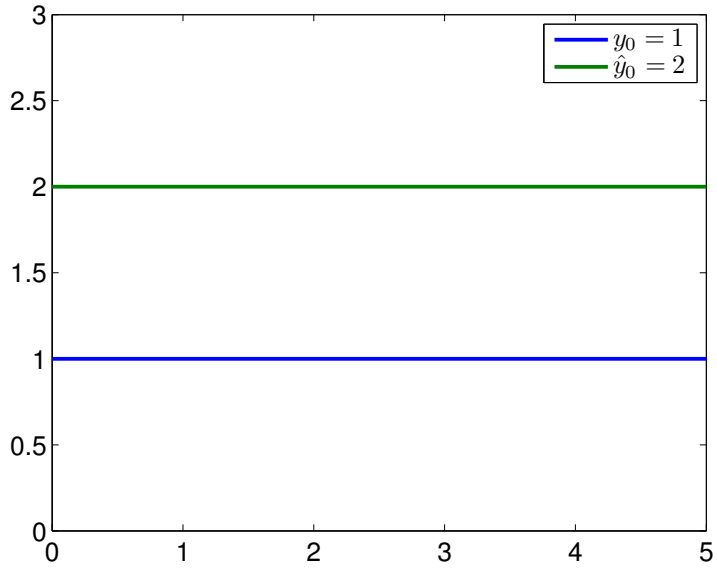
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = -1$, asymptotically stable

ODE Stability

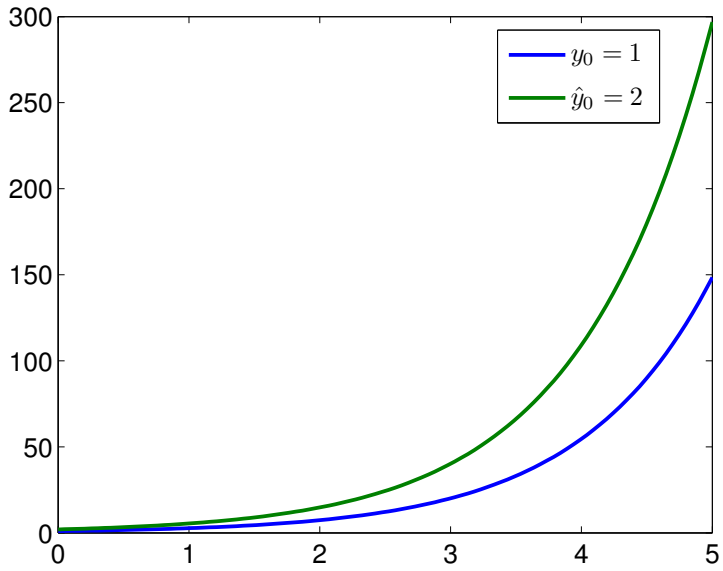
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = 0$, 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[3]

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

Next we consider stability of numerical approximations to ODEs

[3] "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:

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

$$\|\hat{y}_0 - y_0\| \leq \delta \implies \|\hat{y}_k - y_k\| \leq \epsilon$$

for all $k \geq 0$

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 $\mathrm{Re}(\lambda) \leq 0$

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," $\mathrm{Re}(\bar{h}) \leq 0$

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

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:

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

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 $1^2 \leq |1 - (a + ib)|^2$, *i.e.* $(1 - a)^2 + b^2 \geq 1$

Hence, for $\mathrm{Re}(\lambda) \leq 0$, 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 are require less work per time-step since 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 $b = 0$

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

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

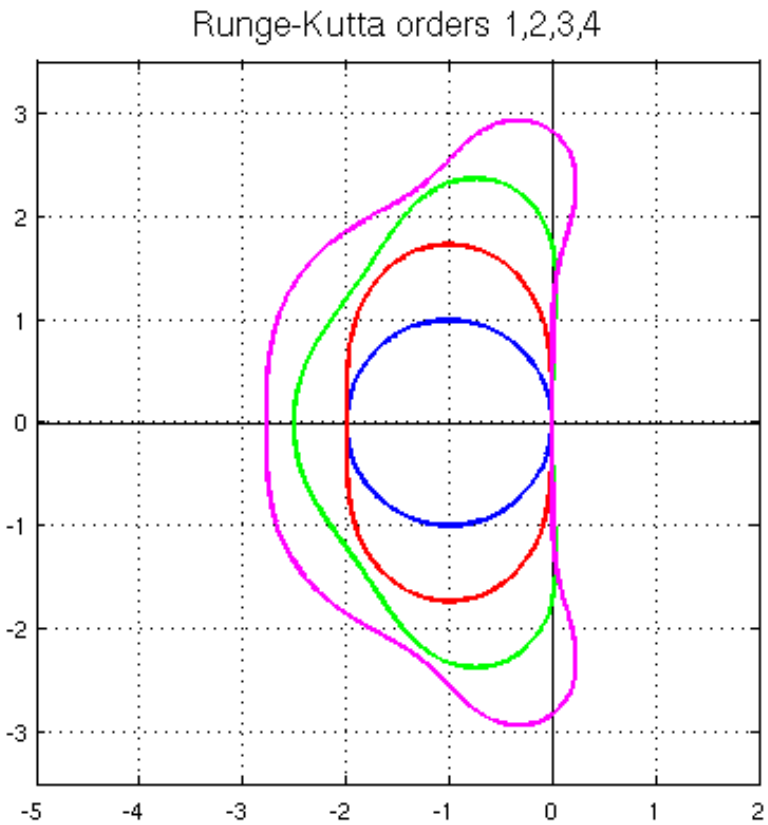
$$\begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + h/2, y_k + hk_1/2) \\ k_3 &= f(t_k + h/2, y_k + hk_2/2) \\ k_4 &= f(t_k + h, y_k + hk_3) \end{aligned}$$

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

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

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

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

Higher-order methods

Fehlberg's 7(8) method[1]

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	
y_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
$\hat{y}_1$	0	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	0	$\frac{41}{840}$	$\frac{41}{840}$

[1] 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[2]

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

[2] 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: 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

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

ODE BVPs

Consider the ODE Boundary Value Problem (BVP):[3] 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

[3] 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:[4] $u'(b) + c_2 u(b) = c_3$

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

ODE BVPs

This is an ODE, so we could try to use the ODE solvers from III.3 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 $x = b$ (just like Angry Birds!)

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 $\gamma U_i \approx \gamma u(x_i)$

Differentiation Matrices

We need a map from the vector $F \equiv [f(x_1), f(x_2), \ldots, f(x_n)] \in \mathbb{R}^n$ to the vector of derivatives $F' \equiv [f'(x_1), f'(x_2), \ldots, f'(x_n)] \in \mathbb{R}^n$

Let $\widetilde{F}'$ denote our finite difference approximation to the vector of derivatives, *i.e.* $\widetilde{F}' \approx F'$

Differentiation is a linear operator[1], hence we expect the map from F to $\widetilde{F}'$ to be an $n \times n$ matrix

This is indeed the case, and this map is a differentiation matrix, D

[1] Since $(\alpha f + \beta g)' = \alpha f' + \beta g'$

Differentiation Matrices

Row i of D corresponds to the finite difference formula for $f'(x_i)$, since then $D_{(i,:)}F \approx f'(x_i)$

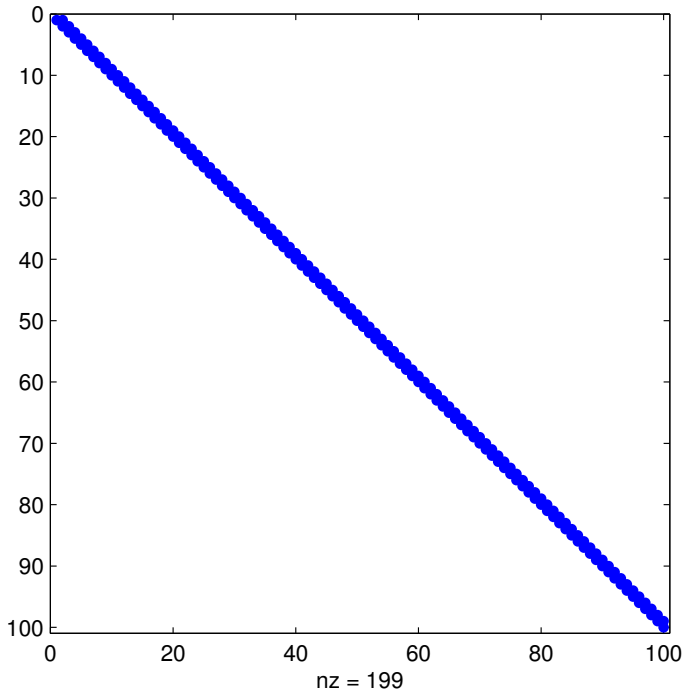
e.g. for forward difference approx. of f', non-zero entries of row i are

$$D_{ii} = -\frac{1}{h}, \quad D_{i,i+1} = \frac{1}{h}$$

This is a sparse matrix with two non-zero diagonals

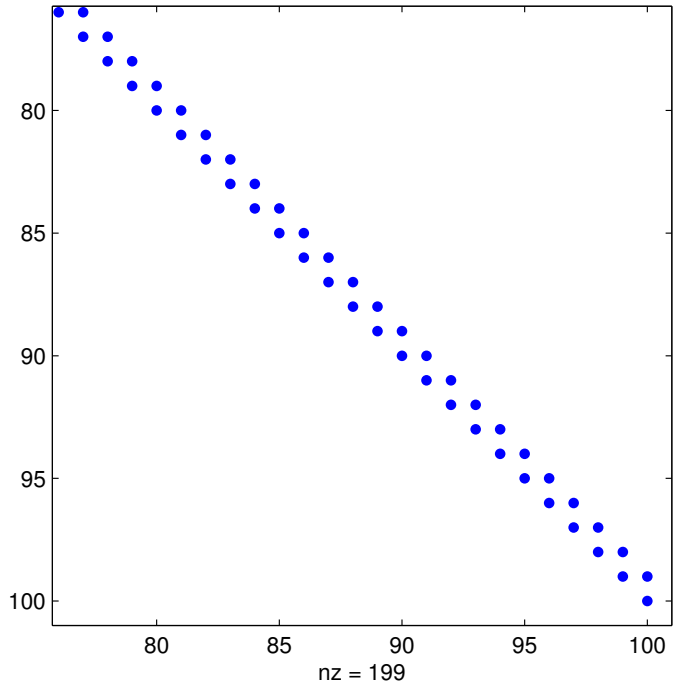
Differentiation Matrices

```
n=100
h=1/(n-1)
D=np.diag(-np.ones(n)/h)+np.diag(np.ones(n-1)/h,1)
plt.spy(D)
plt.show()
```



Differentiation Matrices

But what about the last row?

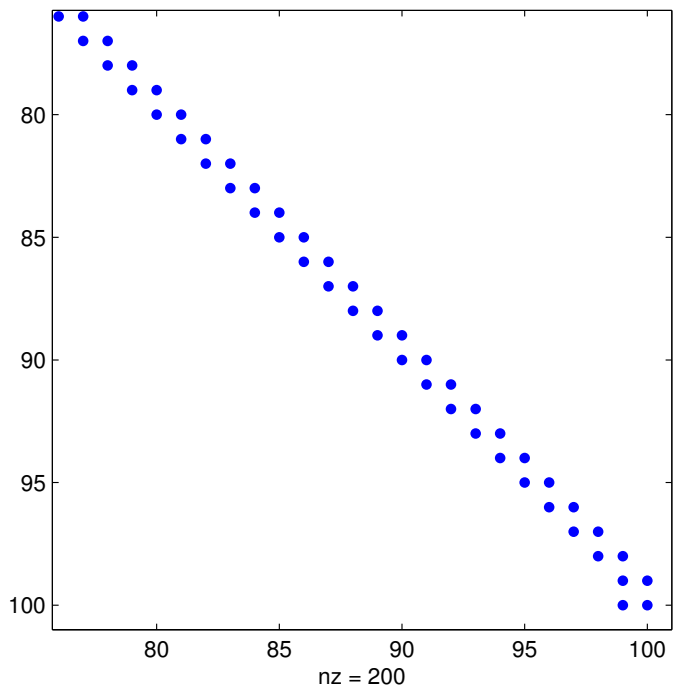


$D_{n,n+1} = \frac{1}{h}$ is ignored!

Differentiation Matrices

We can use the backward difference formula (which has the same order of accuracy) for row n instead

$$D_{n,n-1} = -\frac{1}{h}, \quad D_{nn} = \frac{1}{h}$$



Python demo: Differentiation matrices

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[1] 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!

[1] 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[2] 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 $u(x) = e^x \sin(2\pi x)$:

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

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

ODE BVPs

Python example: ODE BVP via finite differences

Convergence results:

h	error
2.0e-2	5.07e-3
1.0e-2	1.26e-3
5.0e-3	3.17e-4
2.5e-3	7.92e-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 difference formula for $u'(b)$ in the Robin BC, as in III.2