Applied Mathematics 205

Unit III: Numerical Calculus

Lecturer: Dr. David Knezevic

Unit III: Numerical Calculus

Chapter III.3: ODE Initial Value Problems

ODE IVPs

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

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

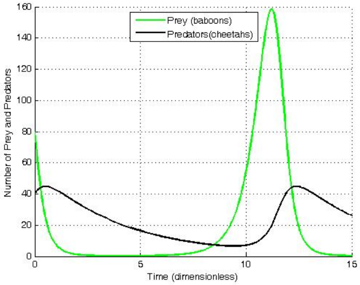
and $y(0) = y_0$, $v(0) = v_0$

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 Matlab

Matlab has many very good ODE IVP solvers

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

Most popular function is `ode45`, which uses the fourth-order Runge-Kutta method (more on this method shortly)

Matlab example: `ode45` for Lotka-Volterra system

In the remainder of this chapter we will discuss how ODE solvers (like `ode45`) work

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 approximation 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 the notation for this, and all methods considered in this chapter, is the same regardless of whether y_k is a vector or a scalar

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

Euler's Method

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

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

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:

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

Matlab example: Euler's method for $y' = \lambda y$

Backward Euler Method

We can derive other ODE solvers just by modifying the quadrature rule

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:

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

Implicit vs. Explicit Methods

(Forward) Euler 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:

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

Implicit vs. Explicit Methods

For example, consider approximation of $y' = 2\sin(ty)$ using backward Euler

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

$$y_1 = y_0 + 2h\sin(t_1 y_1)$$

To compute y_1, let $F(y_1) \equiv y_1 - y_0 - 2h\sin(t_1 y_1)$ 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

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

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 want the global error to converge to 0 as $h \to 0$

We use the truncation error, T_k, to understand the global error

The truncation error is defined as the amount "left over" at step k when we apply our method to the exact solution and divide by h

Convergence

For example, for an explicit one-step ODE approximation, we have:

$$\frac{y_{k+1} - y_k}{h} - \Phi(t_k, y_k; h) = 0,$$

Hence the truncation error in this case is:[2]

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

[2] In some texts truncation error is defined as hT_k

Convergence

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

Suppose there is no error at $t = t_k$, i.e. $y_k = y(t_k)$, then:

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

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

This suggests that 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

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, see II.2

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 $|T_k| = O(h^p)$

Order of accuracy of an ODE solver is generally an integer, and 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

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

Therefore, we have:

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

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 $(b-a)^3 = (t_{k+1} - t_k)^3 = h^3$ (see III.2), hence

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

Order of Accuracy

See lecture: Can also show second order accuracy of trapezoid rule via Taylor expansion

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

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

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

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

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 terminology is a bit confusing compared to previous Units:

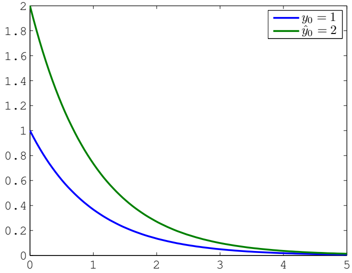
- We previously referred to this type of property as the conditioning of the problem
- Stability previously referred only to properties of a numerical approximation (not of the underlying mathematical problem)

Nevertheless, this is the standard terminology for ODEs (and PDEs)

ODE Stability

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

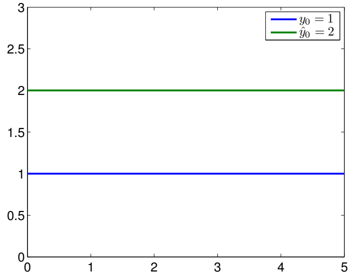
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = -1$, asymptotically stable

ODE Stability

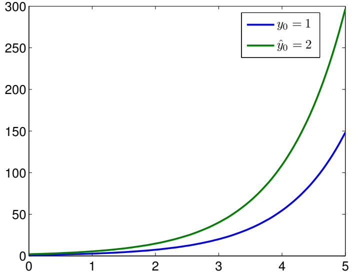
$$y(t) - \hat{y}(t) = (y_0 - \hat{y}_0)e^{\lambda t}$$



$\lambda = 0$, stable

ODE Stability

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



$\lambda = 1$, unstable

ODE Stability

More generally, in the ODE $y' = \lambda y$ we can allow λ to be a complex number: $\lambda = a + ib$

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

It follows that $|y(t) - \hat{y}(t)| = |y_0 - \hat{y}_0| e^{at}$

Hence 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: Systems

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

- $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$, 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: Systems

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:
$\mathrm{Re}(\lambda_i) \leq 0$ for $i = 1, \ldots, n \implies y' = Ay$ is a stable ODE

Next we consider stability of numerical approximations to ODEs

[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:

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

To assess the stability properties of ODE discretizations, 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

It follows that $\|y_k - \hat{y}_k\| = |1 + h\lambda|^k \|y_0 - \hat{y}_0\|$

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

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

For example:

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

In this case λ "more negative" requires tighter restriction on h for stability

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, Backward Euler discretization of $y' = \lambda y$, $\mathrm{Re}(\lambda) < 0$ is stable for any $h > 0$

As a result we say that the backward Euler method is unconditionally stable

Matlab example: `euler_stability_comparison.m`

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

Instead of detailing this framework, we will just 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

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

Two 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) ($a = b = 1/2$, $\alpha = \beta = 1$):

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

Runge-Kutta Methods

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

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

where

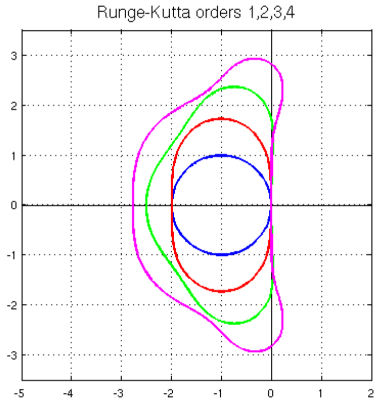
$$\begin{aligned} k_1 &= f(t_k, y_k) \\ k_2 &= f(t_k + h/2, y_k + hk_1/2) \\ k_3 &= f(t_k + h/2, y_k + hk_2/2) \\ k_4 &= f(t_k + h, y_k + hk_3) \end{aligned}$$

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)[7]



[7]Regions computed using the Chebfun toolbox for Matlab

Stiff systems

You may have heard of “stiffness” in the context of ODEs: an important (though somewhat imprecise) concept

Key idea of stiffness: ODE solution has very different timescales

This occurs frequently in practice, e.g. in ODEs modeling chemical reactions

Stiff systems are difficult to solve numerically since we have to resolve “short” and “long” timescales simultaneously

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 very beneficial to use an implicit method since that enforces stability regardless of timestep size

Stiff systems

We say a linear ODE system $y' = Ay$ is stiff if A has eigenvalues that differ greatly in magnitude[8]

Matlab 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

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

[8]You may recall from a "diff. eq." course that eigenvalues of A determine the "timescales" in the solution, e.g. see Matlab example

Stiff systems

Matlab provides ODE solvers that are optimized for stiff systems

These provide adaptive time-stepping, which can be very beneficial for stiff ODEs

Matlab example[9]: `ode15s` vs `ode45` for a stiff problem

[9] From Mathworks website

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

See lecture: Derivation of second order Adams-Bashforth method

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 vs. multi-step methods:

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