Chapter 2 - Initial-Value Problems

2.1 Introduction

Numerical methods for IVPs are distinguished by whether or not they use previous values $\mathbf{y}_{n-1}$, $\mathbf{y}_{n-2}$, etc., along with the current value $\mathbf{y}_n$ in order to compute $\mathbf{y}_{n+1}$.

If they do not, they are called *one-step methods*; otherwise they are called *multi-step methods*[1].

In this course we will only study the popular *linear* multi-step methods. We will see that this only means that the methods are linear functions of the coefficients, not that the IVP is linear.

IVPs can be *stiff* or *non-stiff*.

Stiffness is about efficiency; i.e., it is a computer science thing. Correctly distinguishing between stiff and non-stiff IVPs may be important when choosing a numerical method for their solution.

[1] In the text they are also referred to as *methods with memory*.

No entirely satisfactory definition of stiffness exists in a mathematical sense, although its symptoms may be easy to recognize, e.g., Δt restricted by stability, not accuracy; "widely varying time constants"; etc.

My pragmatic definition of stiffness is that *an instance of a problem is stiff if it is more efficient to obtain a numerical solution by using a stiff solver rather than a non-stiff solver*.

By "instance" of a problem, I mean a given IVP (i.e., ODEs, ICs, time interval) **plus** a given order of method and tolerance.

One (perhaps) unintuitive conclusion of this definition is that *in principle, no IVP is stiff if the tolerance is small enough*.

A general approach to solving IVPs (in MATLAB) is to start with a non-stiff solver (like `ode45`). This solver uses a one-step method based on an embedded pair of explicit Runge–Kutta (RK) formulas (of orders 4 and 5) popularized by Dormand and Prince.

If this proves unsatisfactory, or if you had reason to believe the IVP was stiff to begin with, it is then recommended to try `ode15s`. This solver uses a multi-step solver based on *numerical differentiation formulas* (NDFs) of orders 1 through 5.

NDFs are modifications of the classical *backward differentiation formulas* (BDFs). The modifications are to make the implementation more efficient. It is possible to run `ode15s` as a variable-stepsize, variable-order BDF method by setting option `'BDF'` to `'On'`.

If the IVP has a smooth and/or expensive right-hand side function $\mathbf{f}(t, \mathbf{y})$, then the solver `ode113` is often more efficient than `ode45`. This solver uses multi-step *predictor-corrector* methods based on the family of *Adams methods* of orders 1 through 13.

Solvers known as *type-insensitive solvers* also exist (although not in MATLAB) that attempt to diagnose stiffness internally and choose the appropriate type of solver automatically. These solvers can be particularly effective because they can switch back and forth from stiff to non-stiff solvers as the problem dictates.

2.2 Numerical Methods for IVPs

A standard result from ODE theory is that if $\mathbf{y}_1(t)$ and $\mathbf{y}_2(t)$ are solutions to

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}), \tag{1}$$

and if $\mathbf{f}$ satisfies a Lipschitz condition with constant L, then for $t_1 < t_2$, we have

$$\|\mathbf{y}_1(t_2) - \mathbf{y}_2(t_2)\| \leq \|\mathbf{y}_1(t_1) - \mathbf{y}_2(t_1)\| e^{L(t_2 - t_1)}.$$

The classical situation is that $L(b-0)$ is of moderate size. This tells us that the IVP consisting of (1) with

$$\mathbf{y}(0) = \mathbf{y}_0$$

is moderately stable.

However, **this is only a sufficient condition**: stiff problems are (very) stable, yet $L(b-0) \gg 1$.

A stiff problem is very stable in the sense that solutions of the ODE that start *near* the solution to the IVP converge to it very rapidly; i.e., over a distance that is small compared to the interval of integration.

So some solutions change very rapidly, yet the solution of interest varies slowly (leading to the concept of "multiple time scales", stiffness ratios, etc.).

Finally recall that although numerical methods approximate the solution on a mesh, all of the MATLAB solvers are supplemented with (inexpensive) interpolants $\mathbf{S}(t)$ for approximating the solution between mesh points.

As we will see, BDF methods are based on polynomial interpolation anyway, so it is natural to obtain continuous, piecewise-polynomial interpolants.

Things are less natural in the case of explicit RK (ERK) methods. The continuous, piecewise-polynomial interpolants in this case are also referred to as *continuous extensions* of the RK formula.

2.2.1 One-step methods

One-step methods use only information from the current step.

We focus on *explicit Runge–Kutta* (ERK) methods.

Along the way we develop some implicit Runge–Kutta (IRK) methods.

Of course IRK methods can be used for solving stiff IVPs; however they are also the method of choice for solving BVPs.

We first take up the special case of the *quadrature* problem.

Strictly speaking, *quadrature* is an old-fashioned word that refers to the numerical approximation of definite integrals.

Let $f(x)$ be a real-valued function of a real variable, defined on a finite interval $a \leq x \leq b$.

We seek to compute the value of the definite integral

$$\int_a^b f(x)\,dx.$$

Note: This is a real number.

In the context of the initial-value problem, we have

$$\dot{\mathbf{y}} = \mathbf{f}(t), \quad \mathbf{y}(0) = \mathbf{y}_0.$$

The local solution at t_n satisfies

$$\dot{\tilde{\mathbf{y}}} = \mathbf{f}(t), \quad \tilde{\mathbf{y}}(t_n) = \mathbf{y}_n.$$

This connects to the classical quadrature problem via

$$\tilde{\mathbf{y}}(t_n + \Delta t) = \mathbf{y}_n + \int_{t_n}^{t_n+\Delta t} \mathbf{f}(t)\,dt.$$

Thus computing $\mathbf{y}_{n+1} \approx \tilde{\mathbf{y}}(t_n+\Delta t)$ requires numerical approximation to a definite integral.

A basic tactic in numerical analysis

Replace any "difficult"[2] function $f(t)$ with an interpolating polynomial $P(t)$ and use that instead.

The idea is that polynomials are easy to deal with.

For example, approximate $f(t)$ on $[t_n, t_n + \Delta t]$ by $P(t) = f(t_n)$ (a constant interpolating polynomial).

It is easy to integrate $P(t)$ to obtain

$$\int_{t_n}^{t_n+\Delta t} f(t)\,dt \approx \int_{t_n}^{t_n+\Delta t} P(t)\,dt = \Delta t f(t_n).$$

This is known as the *left-hand rectangle rule*.

[2] This could mean anything from "complicated" to "unknown".

Similarly we could use a constant interpolating polynomial $P(t) = f(t_n + \Delta t)$ to obtain

$$\int_{t_n}^{t_n+\Delta t} f(t)\,dt \approx \int_{t_n}^{t_n+\Delta t} P(t)\,dt = \Delta t f(t_n + \Delta t).$$

This is known as the *right-hand rectangle rule*.

It turns out that the optimal constant interpolating polynomial is $P(t) = f(t_n + \Delta t/2)$, leading to

$$\int_{t_n}^{t_n+\Delta t} f(t)\,dt \approx \int_{t_n}^{t_n+\Delta t} P(t)\,dt = \Delta t f(t_n + \Delta t/2).$$

This is known as the *mid-point rule*.

All of these schemes approximate the integral by the area of a rectangle.

The next generalization from here would be to use a *linear interpolant* $P(t)$. The most natural one interpolates $f(t)$ at both end points:

$$P(t) = \left(\frac{(t_n + \Delta t) - t}{\Delta t}\right) f(t_n) + \left(\frac{t - t_n}{\Delta t}\right) f(t_n + \Delta t).$$

Integrating this $P(t)$ leads to the approximation

$$\int_{t_n}^{t_n+\Delta t} f(t)\, dt \approx \frac{\Delta t}{2}[f(t_n) + f(t_n + \Delta t)].$$

Standard interpolation theory tells us the accuracy of these approximations for smooth $f(t)$.

If $P(t)$ is the unique polynomial of degree less than s that interpolates a smooth function $f(t)$ at s distinct nodes[3] $t_{n,i} := t_n + c_i \Delta t \in [t_n, t_n + \Delta t]$, i.e.;

$$P(t_{n,i}) = f(t_{n,i}), \quad i = 1, 2, \ldots, s,$$

[3] These are also sometimes called *abscissae*.

then $P(t)$ is given by the Lagrange form

$$P(t) = \sum_{i=1}^{s} f_{n,i} \prod_{\substack{j=1 \\ j \neq i}}^{s} \frac{t - t_{n,j}}{t_{n,i} - t_{n,j}},$$

and for all $t \in [t_n, t_n + \Delta t]$, there is a $\hat{t} \in [t_n, t_n + \Delta t]$ such that

$$f(t) - P(t) = \frac{f^{(s+1)}(\hat{t})}{(s+1)!} \prod_{j=1}^{s} (t - t_{n,j}).$$

If $f(t)$ is sufficiently smooth, $|f^{(s+1)}(\hat{t})|$ is bounded, hence there is a constant C such that

$$|f(t) - P(t)| \leq C(\Delta t)^s, \quad \text{for all } t \in [t_n, t_n + \Delta t].$$

We express this as

$$f(t) = P(t) + \mathcal{O}((\Delta t)^s).$$

Now it is easy to see that

$$\int_{t_n}^{t_n+\Delta t} f(t)\,dt = \int_{t_n}^{t_n+\Delta t} P(t)\,dt + \mathcal{O}((\Delta t)^{s+1}).$$

Hence for the left-hand rectangle rule

$$y_{n+1} = y_n + \Delta t f(t_n),$$

we have a local error

$$\tilde{y}(t_n + \Delta t) - y_{n+1} = \mathcal{O}((\Delta t)^2).$$

Similarly, for the right-hand rectangle rule

$$y_{n+1} = y_n + \Delta t f(t_n + \Delta t),$$

we have a local error of the same order:

$$\tilde{y}(t_n + \Delta t) - y_{n+1} = \mathcal{O}((\Delta t)^2).$$

For the trapezoidal rule,

$$y_{n+1} = y_n + \frac{\Delta t}{2}[f(t_n) + f(t_n + \Delta t)],$$

we have a local error

$$\tilde{y}(t_n + \Delta t) - y_{n+1} = \mathcal{O}((\Delta t)^3).$$

For general polynomial interpolation at s nodes, the *interpolatory quadrature formula*

$$\int_{t_n}^{t_n+\Delta t} f(t)\,dt = \Delta t \sum_{i=1}^{s} b_j f(t_n + c_i \Delta t) + \mathcal{O}((\Delta t)^{p+1})$$

leads to a numerical method

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_j f(t_n + c_i \Delta t)$$

with local error $\mathcal{O}((\Delta t)^{p+1})$, where interpolation theory guarantees us that $p \geq s$. [4]

For example, it can be shown that the midpoint rule has local error $\mathcal{O}((\Delta t)^3)$, which is larger than you might expect on general grounds.

[4] This is possible for special choices of the nodes c_i.

Order conditions for quadrature formulas

To determine how accurate a given quadrature formula is, we expand $\tilde{y}(t+\Delta t)$ and y_{n+1} in Taylor series about t_n and see how many terms agree:

$$\tilde{y}(t_n + \Delta t) = \tilde{y}(t_n) + \sum_{i=1}^{p} (\Delta t)^i \frac{\tilde{y}^{(i)}(t_n)}{i!} + \mathcal{O}((\Delta t)^{p+1})$$

$$= y_n + \sum_{i=1}^{p} (\Delta t)^i \frac{f^{(i-1)}(t_n)}{i!} + \mathcal{O}((\Delta t)^{p+1}).$$

Now we use

$$f(t_n + c_i \Delta t) = \sum_{j=0}^{p-1} (c_i \Delta t)^j \frac{f^{(j)}(t_n)}{j!} + \mathcal{O}((\Delta t)^p).$$

Using this result,

$$y_{n+1} = y_n + \Delta t \sum_{i=1}^{s} b_i \left(\sum_{j=1}^{p} (c_i \Delta t)^{j-1} \frac{f^{(j-1)}(t_n)}{(j-1)!} \right) + \mathcal{O}((\Delta t)^{p+1})$$

$$= y_n + \sum_{j=1}^{p} (\Delta t)^j \left(\sum_{i=1}^{s} b_i c_i^{j-1} \right) \frac{f^{(j-1)}(t_n)}{(j-1)!} + \mathcal{O}((\Delta t)^{p+1}).$$

So we see that $\tilde{y}(t_n + \Delta t) = y_{n+1} + \mathcal{O}((\Delta t)^{p+1})$ if and only if

$$\frac{1}{j} = \sum_{i=1}^{s} b_i c_i^{j-1}, \quad j = 1, 2, \ldots, p. \tag{2}$$

Equations (2) are known as the *order conditions* for quadrature (or the *quadrature conditions*).

They are simple algebraic expressions that can be used to check the order of accuracy of a given quadrature formula without the pain of doing Taylor expansions.

Using 1 node ($s = 1$), for $j = 1$ we see that $1 = b_1$.

For $j = 2$, we have

$$\frac{1}{2} = b_1 c_1.$$

Thus if $c_1 \neq 1/2$, the best that we can do is to have $\tilde{y}(t_n + \Delta t) = y_{n+1} + \mathcal{O}((\Delta t)^2)$.

Choosing $c_1 = 1/2$, for $j = 3$ we find

$$\frac{1}{3} \neq 1 \times \left(\frac{1}{2}\right)^2.$$

Hence if $c = 1/2$, $\tilde{y}(t_n + \Delta t) = y_{n+1} + \mathcal{O}((\Delta t)^3)$, and this is the best that can be done.

Convergence of quadrature formulas

We first establish the stability of the quadrature ODE.

If $\mathbf{f} = \mathbf{f}(t)$, then it is easy to see that $\mathbf{f}$ satisfies a Lipschitz condition with $L = 0$.

Thus our previous result on distances between neighbouring solutions says that if $\mathbf{y}_1(t)$ and $\mathbf{y}_2(t)$ are solutions of $\dot{\mathbf{y}} = \mathbf{f}(t)$ and $t_1 < t_2$ then

$$\|\mathbf{y}_1(t_2) - \mathbf{y}_2(t_2)\| \leq \|\mathbf{y}_1(t_1) - \mathbf{y}_2(t_1)\|.$$

However it is easy to prove a stronger result.

A solution $\mathbf{y}(t)$ of $\dot{\mathbf{y}} = \mathbf{f}(t)$ satisfies

$$\mathbf{y}(t) = \mathbf{y}(t_1) + \int_{t_1}^{t} \mathbf{f}(t')\, dt'.$$

Letting $\mathbf{y} = \mathbf{y}_1,\ \mathbf{y}_2$ and $t = t_1,\ t_2$, and subtracting,

$$\mathbf{y}_1(t_2) - \mathbf{y}_2(t_2) = \mathbf{y}_1(t_1) - \mathbf{y}_2(t_1);$$

i.e., *the local error of a step from t_n moves us to a solution of the ODE that is parallel to the solution through $\mathbf{y}_n$.*

Let the true error at t_n be

$$\mathbf{e}_n = \mathbf{y}(t_n) - \mathbf{y}_n.$$

Of course, $\mathbf{y}_0 = \mathbf{y}(0)$ implies that $\mathbf{e}_0 = \mathbf{0}$.

In Chapter 1, we wrote that in general

$$\mathbf{e}_{n+1} = \mathbf{y}(t_{n+1}) - \tilde{\mathbf{y}}(t_{n+1}) + \tilde{\mathbf{y}}(t_{n+1}) - \mathbf{y}_{n+1}.$$

The second difference is the local error that we assume is bounded in magnitude by $C(\Delta t_n)^{p+1}$.

We assume the first difference can be bounded by

$$\|\mathbf{y}(t_{n+1}) - \tilde{\mathbf{y}}(t_{n+1})\| \leq \|\mathbf{y}(t_n) - \tilde{\mathbf{y}}(t_n)\| e^{L\Delta t_n}.$$

Putting these together,

$$\|\mathbf{e}_{n+1}\| \leq C(\Delta t_n)^{p+1} + \|\mathbf{e}_n\| e^{L\Delta t_n}.$$

Thus we see that the error contains two components, one that is the local error introduced at each step by the numerical method and the other that is propagation of error from past steps by the ODE stability.

For quadrature problems, life is simpler because there is no amplification of past errors, and local errors simply add up when computing the bound.

So if we solve a quadrature problems with a constant $\Delta t = (b-0)/n$, we have a *uniform bound*

$$\|\mathbf{e}_n\| \leq nC(\Delta t)^{p+1} \leq (b-0)C(\Delta t)^p;$$

i.e.,

$$\mathbf{y}_n = \mathbf{y}(t_n) + \mathcal{O}((\Delta t)^p), \quad \text{for all } n.$$

This is why we say the order of a method is p when the local error is $\mathcal{O}((\Delta t)^{p+1})$.

For variable step sizes, it is easy to modify the proof to show that the global error is $\mathcal{O}((\Delta T)^p)$ where $\Delta T = \max \Delta t_n$.

Local error estimation

The local error associated with y_{n+1} of a method of order p is

$$le_n = \tilde{y}(t_n + \Delta t) - y_{n+1}.$$

If we also apply a method of order $p+1$ to obtain y^*_{n+1}, we notice

$$\begin{aligned} est &:= y^*_{n+1} - y_{n+1} \\ &= [y^*_{n+1} - \tilde{y}(t_n + \Delta t)] + [\tilde{y}(t_n + \Delta t) - y_{n+1}] \\ &= le_n + \mathcal{O}((\Delta t)^{p+2}); \end{aligned}$$

i.e., est is a **computable, leading-order estimate** of the local error le_n.

Of course Δt needs to be small enough for this to hold.

In other words, we can estimate the error in y_{n+1} by comparing it to the more accurate y^*_{n+1}.

Generally the most expensive part of taking a step is evaluating f.

So *the trick to making local error estimation practical is to have a pair of methods that share as many of these function evaluations as possible*.

Note that we have assumed y^*_{n+1} is the more accurate approximation.

If this is the case, why not use it instead of y_{n+1} to advance the integration?

Advancing with (the more accurate) y^*_{n+1} is called *local extrapolation*; most modern codes do this.

Note however that our error estimate was for y_{n+1}, not for y^*_{n+1}.

This means we do not really know what the error is for y^*_{n+1}, but we assume it is less than est.

IVP solvers control the estimated local error: a tolerance τ is specified, and if $est > \tau$, the step is rejected and a smaller one is tried.

We can see how to adjust the step size by writing out another term in the expansion of the local error:

$$le_n := (\Delta t)^{p+1}\phi(t_n) + \mathcal{O}((\Delta t)^{p+2}),$$

where we have written $\phi(t_n)$ to be more explicit than simply C.

Now suppose we were to take a step from t_n of size $\sigma\Delta t$. Then the local error would be

$$\begin{aligned}(\sigma\Delta t)^{p+1}\phi(t_n) + \mathcal{O}((\sigma\Delta t)^{p+2}) &= \sigma^{p+1}(\Delta t)^{p+1}\phi(t_n) + \mathcal{O}((\Delta t)^{p+2}) \\ &= \sigma^{p+1}est + \mathcal{O}((\Delta t)^{p+2}).\end{aligned}$$

We can now predict the largest Δt that will pass the error test: simply choose σ such that $|\sigma^{p+1}est| \approx \tau$; i.e.,

$$\Delta t_{\text{new}} := \sigma\Delta t = \Delta t\left(\frac{\tau}{|est|}\right)^{\frac{1}{p+1}}.$$

The solver will keep trying to find a step size for which $|est| \leq \tau$ until it succeeds or gives up.

It may give up because it has done too much work; more often it will give up if it predicts that it will need a step size that is too small for the precision of the computer. This is a similar situation to asking for too much accuracy.

However, a successful step may estimate that the local error is smaller than necessary; hence we might think about taking a *larger* step size for the next step.

This increases efficiency!

It turns out that we can use the above estimate for σ to also tell us what Δt to choose for the next step:

$$\begin{aligned}(\sigma\Delta t)^{p+1}\phi(t_{n+1}) + \mathcal{O}((\sigma\Delta t)^{p+2}) &= \sigma^{p+1}(\Delta t)^{p+1}\phi(t_n) + \mathcal{O}((\Delta t)^{p+2}) \\ &= \sigma^{p+1} est + \mathcal{O}((\Delta t)^{p+2}).\end{aligned}$$

This is generally how modern codes choose their step sizes, but there are various "practical" details missing.

For example, the range of allowable σ must be restricted in practice because est will not be reliable if Δt changes by a lot.

Also, several approximations are made in deriving the formula for σ, so it should not be taken too seriously.

In particular, the value actually used by the software is actually $\alpha\sigma$, where α is a fudge factor of about 0.9.

This is meant to be conservative.

Rejected steps are expensive, so we wish to avoid these if at all possible.

This conservative strategy is to obtain the lion's share of the benefit while reducing the risk of failure.

For those who are familiar with digital control, there are step size controllers based on classical control theory (i.e., proportional-integral-derivative (PID) control).

Estimation and control of the local error are of critical importance in practice: Without them, we have no way of knowing that we have computed a meaningful solution to an IVP.

Estimation and control of the local error do not have to be expensive!

The minor added expense more than pays for itself in increased step sizes: constant step-size strategies are worst-case strategies because they are restricted by the smallest step sizes required to resolve the problem or maintain stability anywhere the entire interval.

For example, the proton transfer problem of Section 1.4 has a boundary layer of approximate width 10^{-10}.

To resolve this we need a step size on this order.

But the problem is posed on a time interval $[0, 10^6]$.

So with a constant step size, we would need 10^{17} steps!

Even with a computer capable of 10 GFlops, this computation would still take more than 11.5 days[5], instead of a few seconds on a modest laptop.

Step sizes used by a variable step-size solver in solving this problem range from 7×10^{-14} inside the boundary layer to 4×10^{4} where the solution is slowly varying.

[5] This also excludes the tremendous cumulative effects of round-off errors!

Runge–Kutta methods

For general IVPs, the local solution satisfies

$$\dot{\tilde{\mathbf{y}}} = \mathbf{f}(t, \tilde{\mathbf{y}}), \quad \tilde{\mathbf{y}}(t_n) = \mathbf{y}_n.$$

Integrating this, we obtain

$$\tilde{\mathbf{y}}(t_n + \Delta t) = \mathbf{y}_n + \int_{t_n}^{t_n+\Delta t} \mathbf{f}(t, \tilde{\mathbf{y}}(t))\, dt.$$

The key difference is of course the presence of $\tilde{\mathbf{y}}(\cdot)$ on both sides of the equation.

Approximating the integral with a quadrature formula leads to

$$\int_{t_n}^{t_n+\Delta t} \mathbf{f}(t, \tilde{\mathbf{y}}(t))\, dt = \Delta t \sum_{j=1}^{s} b_j \mathbf{f}(t_{n,j}, \tilde{\mathbf{y}}(t_{n,j})) + \mathcal{O}((\Delta t)^{p+1}).$$

However, we still have the problem that the values $\tilde{\mathbf{y}}(t_{n,j})$ are unknown.

There are 2 basic approaches to dealing with this problem; but first let us consider a few familiar yet important formulas for which this is not a problem.

For the left-hand rectangle rule,

$$\int_{t_n}^{t_n+\Delta t} \mathbf{f}(t, \tilde{\mathbf{y}}(t))\, dt = \Delta t\, \mathbf{f}(t_n, \tilde{\mathbf{y}}(t_n)) + \mathcal{O}((\Delta t)^2)$$
$$= \Delta t\, \mathbf{f}(t_n, \mathbf{y}_n) + \mathcal{O}((\Delta t)^2),$$

leading to the *(forward) Euler (FE) method*:

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

For this method, we have

$$\mathbf{y}_{n+1} = \tilde{\mathbf{y}}(t_n + \Delta t) + \mathcal{O}((\Delta t)^2),$$

so it is a first-order explicit Runge–Kutta method.

The right-hand rectangle rule leads to the *backward Euler (BE) method*:

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

Because the unknown appears on both sides of the equation, this is an implicit Runge–Kutta method.

Of course we did not have such difficulties with quadrature problems!

Also this method is no more accurate than forward Euler, so why would anyone bother with it?

The main reason is to overcome stiffness.

It so happens that backward Euler is also the lowest-order BDF method (BDF1) that we will derive later.

The trapezoidal rule[6]

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

is a second-order IRK method; it is used in `ode23t`.

Because the formula is invariant under the transformation[7]

$$\mathbf{y}_n \to \mathbf{y}_{n+1}, \ \mathbf{y}_{n+1} \to \mathbf{y}_n,$$

$$t_{n+1} \to t_n, \ t_n \to t_{n+1}, \ \Delta t \to -\Delta t,$$

the trapezoidal rule is said to be *symmetric*.

This is particularly useful for *time-reversible flows*; such formulas do not have a preferred direction.

Normally IVPs have a sense of preferred direction dictated by the time variable. BVPs do not, however, and so symmetric formulas are popular BVP solvers.

[6] In the context of PDEs, this method is more commonly known as the *Crank-Nicolson method*.

[7] This transformation reverses the direction of time.

Approximating intermediate values $\mathbf{y}_{n,j}$

Let's suppose we already have a way to obtain

$$\mathbf{y}_{n,j} = \tilde{\mathbf{y}}(t_{n,j}) + \mathcal{O}((\Delta t)^p);$$

i.e., we have local approximations that are of the *global* order of the method.

In other words, we assume there is a constant C such that

$$\|\tilde{\mathbf{y}}(t_{n,j}) - \mathbf{y}_{n,j}\| \leq C(\Delta t)^p.$$

Using the Lipschitz condition that we have assumed on $\mathbf{f}$, we have

$$\begin{aligned}\|\mathbf{f}(t_{n,j}, \tilde{\mathbf{y}}(t_{n,j})) - \mathbf{f}(t_{n,j}, \mathbf{y}_{n,j})\| &\leq L\|\tilde{\mathbf{y}}(t_{n,j}) - \mathbf{y}_{n,j}\| \\ &\leq LC(\Delta t)^p.\end{aligned}$$

So if we use (the computable) $\mathbf{f}(t_{n,j}, \mathbf{y}_{n,j})$ in place of $\mathbf{f}(t_{n,j}, \tilde{\mathbf{y}}(t_{n,j}))$, we can see

$$\int_{t_n}^{t_n+\Delta t} \mathbf{f}(t, \tilde{\mathbf{y}}(t))\, dt = \Delta t \sum_{i=1}^{s} b_i \mathbf{f}(t_{n,j}, \mathbf{y}_{n,j}) + \mathcal{O}((\Delta t)^{p+1}).$$

Thus if we can use another method to compute intermediate values $\mathbf{y}_{n,j}$ that are (locally) accurate to $\mathcal{O}((\Delta t)^p)$, we can use them in a method of the form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{s} b_i \mathbf{f}(t_{n,j}, \mathbf{y}_{n,j})$$

that will be locally accurate to $\mathcal{O}((\Delta t)^{p+1})$.

Explicit RK methods

ERK methods use the strategy of forming intermediate values by "mini" ERK methods within the larger one.

For example, all ERK methods start with a mini forward Euler step.

This mini FE step has local accuracy $\mathcal{O}((\Delta t)^2)$, so it can be used as an acceptable intermediate value for any quadrature scheme with global accuracy $p = 2$.

For example, the trapezoidal quadrature rule leads to a two-stage, second-order ERK method known as *Heun's method*:

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

Similarly, we can use Heun's method to produce the intermediate values for a quadrature formula to obtain an ERK of order 3, etc.

This leads us to the general form of the ERK method:

$$\mathbf{y}_{n,i} = \mathbf{y}_n + \Delta t \sum_{j=1}^{i-1} a_{ij} \mathbf{f}_{n,j}, \quad i = 1, 2, \ldots, s,$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{s} b_i \mathbf{f}_{n,i},$$

where

$$\mathbf{f}_{n,j} := \mathbf{f}(t_n + c_j \Delta t, \mathbf{y}_{n,j}).$$

Note 1. *The sum over j is empty when $i = 1$.*

Note 2. *You will often see $\mathbf{f}_{n,i}$ simply written as $\mathbf{K}_i$.*

If we translate the quadrature order conditions to this notation, we find that a method will have (global) order p (for quadrature problems) if and only if

$$\frac{1}{j} = \sum_{i=1}^{s} b_i c_i^{j-1}, \quad \text{for all } j = 1, 2, \ldots, p.$$

These will certainly be *necessary conditions* for a general ERK method to have order p, but they are **not sufficient** because in general $\mathbf{f} = \mathbf{f}(t, \mathbf{y})$, not $\mathbf{f} = \mathbf{f}(t)$ as it was for quadrature problems.

Sadly it is (likely!) beyond the scope of this course to go into the details of a full derivation of the order conditions for RK methods.

Here are the number of order conditions τ as a function of order p:

p	1	2	3	4	5	6	7	8	9	10
τ	1	2	4	8	17	37	85	200	486	1205

As we can see τ grows combinatorially with p.

The Butcher tableau

The coefficients defining a Runge–Kutta method are conveniently presented in tableau form, a notation due to Butcher:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$$

It is a common assumption that

$$\sum_{j=1}^{s} a_{i,j} = c_i, \qquad i = 1, 2, \ldots, s,$$

This ensures that the time variable is treated consistently.

With this assumption we can then make the interpretation that

$$\mathbf{y}_{n,i} \approx \mathbf{y}(t_n + \Delta t\, c_i).$$

For an ERK, $\mathbf{A}$ is *strictly lower-triangular*: all entries on and above the diagonal are 0 (and are usually not displayed).

An entire theory in terms of "rooted trees" exists (and was originally developed by John Butcher) that gives the order conditions for a general RK method in terms of its coefficients.

For example, here are the general RK order conditions up to order 3:

Order 1: $$\sum_{i=1}^{s} b_i = 1$$

Order 2: $$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}$$

Order 3: $$\sum_{i=1}^{s} b_i c_i^2 = \frac{1}{3}, \quad \sum_{i,j=1}^{s} b_i a_{ij} c_j = \frac{1}{3!}$$

Since then there have been modifications to this theory (most notably by Peter Albrecht) that make these conditions even more palatable, especially to the non-graph theorist.

Here is what those order conditions look like:[8]

$$\begin{aligned}
&\text{Order 1:} && \mathbf{b}^T\mathbf{e} = 1 \\
&\text{Order 2:} && \mathbf{b}^T\mathbf{c} = \frac{1}{2} \\
&\text{Order 3:} && \boldsymbol{\gamma}_2 := \frac{\mathbf{c}^2}{2} - \mathbf{A}\mathbf{c} \\
& && \mathbf{b}^T\mathbf{c}^2 = \frac{1}{3}, \quad \mathbf{b}^T\boldsymbol{\gamma}_2 = \mathbf{0} \\
&\text{Order 4:} && \boldsymbol{\gamma}_3 := \frac{\mathbf{c}^3}{3} - \mathbf{A}\mathbf{c}^2 \\
& && \mathbf{b}^T\mathbf{c}^3 = \frac{1}{4}, \quad \mathbf{b}^T\boldsymbol{\gamma}_3 = \mathbf{0}, \\
& && \mathbf{b}^T\mathbf{A}\gamma_2 = \mathbf{0}, \quad \mathbf{b}^T\mathbf{C}\boldsymbol{\gamma}_2 = \mathbf{0}.
\end{aligned}$$

where $\mathbf{b} = (b_1, b_2, \ldots, b_s)^T$, $\mathbf{e} = (1, 1, \ldots, 1)^T$, $\mathbf{c} = (c_1, c_2, \ldots, c_s)^T$, $\mathbf{C} = \operatorname{diag}(\mathbf{c})$, and expressions like $\mathbf{c}^k$ are understood to apply componentwise.

[8] In reality, they are linear combinations of the other order conditions.

Stages (s) vs. order (p)

Much effort has gone into finding theory for the minimum number of stages (function evaluations) s for a given ERK to have order p.

Here is a summary of some results:

p	1	2	3	4	5	6	7	8
s	1	2	3	4	6	7	9	10

Order 14 seems to be the highest order for which an ERK method has been constructed.

For **linear, constant-coefficient problems**, it is possible to have schemes with $s = p$ for any s, but this is not true for the general problem beyond $s = p = 4$.

However, this consideration may not be as important as it might seem.

Extra stages may allow for a more accurate method, and hence larger steps — perhaps even large enough to offset the extra cost!

Besides, the real issue is in having a pair of formulas that share stages so that the local error can be inexpensively estimated.

Deriving so-called *embedded RK pairs* that share many stages is a challenging task!

We use the following Butcher tableau notation to describe embedded pairs:

$\mathbf{c}$	$\mathbf{A}$
	$\mathbf{b}^T$
	$\hat{\mathbf{b}}^T$

There are a couple of conventions used when talking about embedded pairs of RK methods.

The text[9] prefers to labels the pairs $(p, p+1)$.

For example, the Fehlberg pair from 1970 is denoted $F(4,5)$; the popular pair of Dormand and Prince

[9] and hence MATLAB?

(DOPRI5) from 1980 is the formula used in `ode45`, hence its name.

Note 3. *$F(4,5)$ uses 6 stages, whereas DOPRI5 uses 7 stages. The extra stage is used to design a more accurate formula, and in practice DOPRI5 performs somewhat better than $F(4,5)$.*

There is another convention when it comes to describing embedded pairs.

In this case, there is a distinction between which method is used to advance the integration and which is used to provide the error estimate.

Suppose a pair methods have orders p and $\hat{p}$, where the main method of order p is embedded in the (auxiliary) method of order $\hat{p}$.

Then the embedded pair is denoted as $p(\hat{p})$.

Nowadays, $p = \hat{p} + 1$; i.e., we always do local extrapolation (the main method is of higher order), but in the past this was not always the case.

So for example, the Fehlberg pair $F(4,5)$ really was derived as a $4(5)$ pair, whereas the Dormand–Prince pair DOPRI5 was derived as a $5(4)$ pair.

In other words, Fehlberg did not intend to use local extrapolation, but Dormand and Prince did.

Here is the extended Butcher tableau of the Euler–Heun (1,2) pair:

0		
1	1	
	1	
	$\frac{1}{2}$	$\frac{1}{2}$

Autonomous form

To make the algebra a little simpler, much of the analysis on ODEs is given in terms of the *autonomous ODE* $\mathbf{f} = \mathbf{f}(\mathbf{y})$.

It is always possible to convert a non-autonomous ODE of size m of the form $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ to an autonomous ODE $\dot{\mathbf{Y}} = \mathbf{F}(\mathbf{Y})$ of size $m + 1$ by adding an extra variable that corresponds to time.

Noting $\dot{t} = 1$, we let

$$\mathbf{Y}(t) := (\mathbf{y}^T, t)^T, \qquad \mathbf{F}(\mathbf{Y}) := (\mathbf{f}(t, \mathbf{y})^T, 1)^T.$$

Assuming we have the initial condition $\mathbf{y}(0) = \mathbf{y}_0$, this is translated to $\mathbf{Y}(0) = (\mathbf{y}_0^T, 0)^T$.

Note that this is mainly a technique used for *analysis*; most software packages do not require the user to enter the problem in autonomous form; it is generally more efficient to handle the non-autonomous form directly.

"First Same As Last" (FSAL)

There is a common way to squeeze a little more efficiency out of an embedded ERK pair.

Consider the Bogacki–Shampine 3(2) pair (BS3(2)) used as the integration method in ode23.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

The extended Butcher tableau emphasizes the presence of the abscissa $c_4 = 1$.

The function evaluation associated with this is used with the auxiliary formula to form $\hat{\mathbf{y}}_{n+1}$, which is used to form the error estimate.

However, this is precisely the same function evaluation that is required at the first stage of the next step!

So this information can be saved; it does not have to be recomputed (if the step is accepted), thus saving one function evaluation per step; i.e., this method essentially costs 3 function evaluations per step, not 4.

This is called the "first-same-as-last" (FSAL) property.

The actual algorithm is: Evaluate the 3rd-order result $\mathbf{y}_{n+1}$ from the 3rd stage. This is the approximation at t_{n+1}. The 4th stage uses $\mathbf{f}(t_{n+1}, \mathbf{y}_{n+1})$ to compute $\hat{\mathbf{y}}_{n+1}$, which is differenced from $\mathbf{y}_{n+1}$ to form the error estimate. Set $\mathbf{f}_{n+1,1} = \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1})$; this is the first function evaluation of the next step.

You can imagine that deriving embedded pairs with the FSAL property makes an already difficult task even more so.

The 7-stage Dormand–Prince pair implemented in ode45 has this property; in practice is costs little more than 6 stages per step; this is the minimum required for an ERK to be 5th order anyway.

Continuous extensions

Plotting packages generally just draw straight lines to connect data points.

However, RK formulas of medium to high order take such large steps that joining the output points by straight lines is noticeably poor.

For graphs to be smooth, or in general to obtain solution values between output points that are of the same accuracy as the output points, we need an (inexpensive) interpolant or *continuous extension*.

This is also called the *dense output* feature when talking about RK methods.

Continuous extensions are a relatively recent development for RK methods.

The idea is to use the function evaluations and the intermediate solution values in $[t_n, t_{n+1}]$ to form a polynomial interpolant that has the same order of accuracy as the $\mathbf{y}_{n+1}$.

For example, BS3(2) uses $\mathbf{y}_n$, $\mathbf{f}_n$, $\mathbf{y}_{n+1}$, and $\mathbf{f}_{n+1}$ (all information that is available anyway) to construct a cubic Hermite interpolating polynomial through the solution and its slopes at both end points.

Interpolation theory then guarantees us that the interpolant has the same order of accuracy as $\mathbf{y}_{n+1}$.

This construction on each subinternval yields a piecewise-cubic polynomial $\mathbf{S}(t) \in C^1[0, b]$.

ode45 also has a continuous extension associated with it, without requiring any extra stages; however this continuous extension is only 4th order.

The form of the polynomial interpolant itself is not easy to write down (or look at).

It is expressed as a function of a parameter $\theta \in [0, 1]$ with variable coefficients $b_i(\theta)$, so it looks a lot like the evaluation of $\mathbf{y}_{n+1}$:

$$\mathbf{S}(t_n + \theta\Delta t) = \mathbf{y}_n + \sum_{i=1}^{s^*} b_i(\theta)\mathbf{f}_{n,i},$$

where s^* is the number of stages used to form the interpolant.

Hopefully, $s^* \leq s$ and the stages coincide, but this is not necessarily the case; i.e., extra stages may be required to form the interpolant.

For consistency, $b_i(0) = 0$ and $b_i(1) = b_i$.

2.2.2 Linear Multi-step methods

Once we have reached t_n, we generally have values $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, . . . and slopes $\mathbf{f}_n$, $\mathbf{f}_{n-1}$, . . . that are available for use in computing $\mathbf{y}_{n+1}$.

It is natural to use quadrature here as well.

But now instead of writing

$$\tilde{\mathbf{y}}(t_n + \Delta t) = \mathbf{y}_n + \int_{t_n}^{t_n+\Delta t} \mathbf{f}(t, \tilde{\mathbf{y}}(t))\, dt,$$

interpolating $\mathbf{f}$ at s values $t_{n,j} \in [t_n, t_n + \Delta t]$, and exactly integrating the interpolant, we take $t_{n,j} = t_{n-j}$ because we already have $\mathbf{y}_{n-j}$ and $\mathbf{f}_{n-j}$.

The family of methods so derived is called *Adams–Bashforth* methods.

The first method in this family is denoted AB1, but it is identical to the forward Euler method (it interpolates only $\mathbf{f}_n$).

Interpolating $\mathbf{f}_n$ and $\mathbf{f}_{n-1}$ leads to AB2

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t_n \left[\left(1 + \frac{r}{2}\right) \mathbf{f}_n - \left(\frac{r}{2}\right) \mathbf{f}_{n-1} \right],$$

where $r = \Delta t_n / \Delta t_{n-1}$.

Note 4. *The non-uniform mesh spacing must be taken into account explicitly in the formula; i.e. the coefficients are not fixed!*

For theoretical purposes, Δt is often assumed constant; then AB2 simplifies to

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \left[\frac{3}{2}\mathbf{f}_n - \frac{1}{2}\mathbf{f}_{n-1} \right].$$

Adams–Bashforth methods are explicit linear multi-step methods.

To derive an implicit family of methods, we interpolate using $\mathbf{f}_{n+1}$ as well.

This family of methods is known as *Adams–Moulton* methods.

The first method in this family is denoted AM1, but it is identical to the backward Euler method. It is a one-step method, but it uses only $\mathbf{f}_{n+1}$.

The second member in this family, denoted AM2, is also a one-step method, but it uses $\mathbf{f}_{n+1}$ and $\mathbf{f}_n$. It is identical to trapezoidal rule.

The accuracy of Adams methods can be analyzed in a manner similar to what we did for RK methods.

It turns out that, for a given order k, AMk methods are more accurate and stable than ABk methods.

Which method is preferable for a given problem depends on how difficult (costly) it is to solve the nonlinear equations at each step for the AM methods.

For non-stiff problems, we can do this by *functional iteration*.

Functional iteration

For concreteness, consider AM1: we want to solve the algebraic equations

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

for $\mathbf{y}_{n+1}$.

Suppose we have a guess $\mathbf{y}_{n+1}^{[\nu]}$. Then functional iteration leads to the iteration

$$\mathbf{y}_{n+1}^{[\nu+1]} = \mathbf{y}_n + \Delta t \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]}).$$

This process is known as "updating" or "correcting" the current iterate; hence the implicit formula is often called a *corrector*.

Assuming the algebraic equations have a solution, the Lipschitz condition on $\mathbf{f}$ implies

$$\begin{aligned}\|\mathbf{y}_{n+1} - \mathbf{y}_{n+1}^{[\nu+1]}\| &= \|\Delta t\, \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}) - \Delta t\, \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]})\| \\ &\leq \Delta t\, L \|\mathbf{y}_{n+1} - \mathbf{y}_{n+1}^{[\nu]}\|.\end{aligned}$$

Thus if $\Delta t\, L < 1$, $\mathbf{y}_{n+1}^{[\nu+1]}$ is closer to $\mathbf{y}_{n+1}$ than $\mathbf{y}_{n+1}^{[\nu]}$ was, leading to convergence.

In fact, this argument is a proof that for Δt sufficiently small, the algebraic equations have a solution, and this solution is unique.

Clearly, the convergence rate increases as Δt decreases.

This is important because each iterate costs an evaluation of $\mathbf{f}$, so if we need many iterates, it might make more sense to use an ERK with a smaller step size to achieve the same accuracy.

We want Δt to be as large as we can afford; one important way to reduce the number of iterations is to make a good initial guess $\mathbf{y}_{n+1}^{[0]}$.

An easy and natural way to do this is to use an explicit formula, consequently known as a *predictor*.

For AMk formulas, a natural predictor is ABk or AB$(k-1)$.

Another important way to reduce the cost of the iteration is to note that in practice the iteration only needs to carried out until the accuracy of the integration is not adversely affected, *and not necessarily until the equations are satisfied exactly*.

Note however that functional iteration is only practical for non-stiff problems: if $L(b-0)$ is not large then choosing Δt small enough so that $\Delta t\, L < 1$ will not restrict Δt greatly.

For non-stiff problems, normally the accuracy requirement forces Δt to be small enough to ensure rapid convergence.

Predictor-Corrector Methods

Combining the ideas on how to lower the cost of functional iteration, we come up with an important class of methods known as *predictor-corrector methods*.

The idea here is that a prediction is made with an AB formula, and a *fixed* number of corrections are applied with an AM formula.

In fact, the number of corrections is usually only 1!

Heun's method in fact can be interpreted in this way:

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

This is equivalent to a prediction $\mathbf{y}_{n,2} = \mathbf{y}_{n+1}^{[0]}$ using forward Euler (AB1) followed by one correction using trapezoidal rule (AM2).

Note that we also need to evaluate $\mathbf{f}(t_{n+1}, \mathbf{y}_{n+1})$ for the next step.

This leads to a PECE method (predict-evaluate-correct-evaluate).

You might think it would be better or more natural to use AB2 to predict for AM2, but it is not necessary, and there are advantages to using AB1.

First it is easy to see that a predictor of order $k-1$ plus one corrector iteration of order k leads to a predictor-corrector formula of order k; the leading term is not the same anymore even though its order is.

It is important to realize that a predictor-corrector method is an *explicit* method: despite its use of an implicit formula, the implicit formula is never fully evaluated.

Hence, predictor-corrector methods should only be used for non-stiff problems.

One practical difficulty with implementing implicit methods is deciding on how accurately to solve the implicit equations.

This issue does not apply to predictor-corrector methods because only a fixed number of correction iterations are ever performed; no measure of convergence is present.

BDF Methods

From time t_n, the backward differentiation formula of order k (BDFk) approximates $\mathbf{y}(t_{n+1})$ by the polynomial $\mathbf{P}(t)$ that interpolates $\mathbf{y}_{n+1}$ and the previous k solution values $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, . . . , $\mathbf{y}_{n-k+1}$.

Of course, because we are using this interpolant to define $\mathbf{y}_{n+1}$, we need one more condition to completely specify $\mathbf{P}(t)$.

The final condition is that we force $\mathbf{P}(t)$ to satisfy the ODE at t_{n+1}; i.e.,

$$\dot{\mathbf{P}}(t_{n+1}) := \mathbf{f}(t_{n+1}, \mathbf{P}(t_{n+1})) = \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}).$$

We say that $\mathbf{P}(t)$ *collocates* the ODE at t_{n+1}.

This is an important concept in the solution of differential equations that will come up again in the solution of BVPs.

BDF1 uses a linear polynomial that interpolates the solution at t_n and t_{n+1}.

The interpolant has a constant slope, so the collocation condition reduces to

$$\frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t_n} = \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}),$$

or

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

BDF2 uses a quadratic polynomial interpolant at t_{n-1}, t_n, and t_{n+1}, leading to

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

where $r = \Delta t_n / \Delta t_{n-1}$.

We will see that BDFs are actually used with a constant step size Δt for as many steps as possible; then BDF2 is

$$\frac{3}{2}\mathbf{y}_{n+1} - 2\mathbf{y}_n + \frac{1}{2}\mathbf{y}_{n-1} = \Delta t_n \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}).$$

Adams and BDF methods belong to the class of *linear multi-step methods* (LMMs), which for constant step size are written

$$\sum_{i=0}^{k} \alpha_i \mathbf{y}_{n-i+1} = \Delta t \sum_{i=0}^{k} \beta_i \mathbf{f}_{n-i+1}.$$

For one-step methods, we proved convergence by bounding the propagation of the local error in terms of the stability of the IVP.

Proving convergence of multi-step methods is harder than for one-step methods because the error in the current step depends much more strongly on the errors made in previous steps.

This necessitates a shift in focus from approximating local solutions to approximating the global solution, and from considering the IVP stability to that of the numerical method.

Local Truncation Error

We define the *local truncation error*[10] ($\mathbf{lte}_n$) of a linear multi-step method by

$$\begin{aligned}\mathbf{lte}_n &:= \sum_{i=0}^{k} \alpha_i \mathbf{y}_{n-i+1} - \Delta t \sum_{i=0}^{k} \beta_i \mathbf{f}_{n-i+1} \\ &= \sum_{i=0}^{k} \alpha_i \mathbf{y}_{n-i+1} - \Delta t \sum_{i=0}^{k} \beta_i \dot{\mathbf{y}}(t_{n-i+1}).\end{aligned}$$

Straightforward Taylor series about t_{n+1} reveals that

$$\mathbf{lte}_n = \sum_{j=0}^{\infty} C_j (\Delta t)^j \mathbf{y}^{(j)}(t_{n+1}),$$

[10] also called the *discretization error*

where

$$C_0 = \sum_{i=0}^{k} \alpha_i, \qquad C_1 = \sum_{i=0}^{k} [i\alpha_i + \beta_i],$$

$$C_j = (-1)^j \sum_{i=1}^{k} \left[\frac{i^j \alpha_i}{j!} + \frac{i^{j-1} \beta_i}{(j-1)!} \right], \quad j = 2, 3, \ldots.$$

So a convergent LMM will be of order p if and only if $C_j = 0$ for $j = 0, 1, \ldots, p$, and moreover, we know that

$$\begin{aligned} \mathbf{lte}_n &= C_{p+1}(\Delta t)^{p+1} \mathbf{y}^{(p+1)}(t_{n+1}) + \ldots \\ &= C_{p+1}(\Delta t)^{p+1} \mathbf{y}^{(p+1)}(t_n) + \ldots; \end{aligned}$$

i.e., we have a convenient expression for the leading error coefficient.

Convergence of LMMs

Suppose we have an explicit LMM

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{k} \beta_i \mathbf{f}(t_{n-i+1}, \mathbf{y}_{n-i+1}).$$

The exact solution satisfies

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + \Delta t \sum_{i=1}^{k} \beta_i \mathbf{f}(t_{n-i+1}, \mathbf{y}(t_{n-i+1})) \; + \; \mathbf{lte}_n.$$

Subtracting these two equations, noting the local truncation error is $\mathcal{O}((\Delta t)^{p+1})$, taking norms, and using the Lipschitz condition

$$\|\mathbf{y}(t_{n+1}) - \mathbf{y}_{n+1}\| \leq \|\mathbf{y}(t_n) - \mathbf{y}_n\|$$
$$+\Delta t \sum_{i=1}^{k} |\beta_i| \, L \, \|\mathbf{y}(t_{n-i+1}) - \mathbf{y}_{n-i+1}\| + C(\Delta t)^{p+1}.$$

Notice that there are errors from steps before t_n.

To deal with them, we define

$$E_n := \max_{j \le n} \|\mathbf{y}(t_j) - \mathbf{y}_j\|.$$

Then,

$$\|\mathbf{y}(t_{n+1}) - \mathbf{y}_{n+1}\| \le (1 + \Delta t \mathcal{L}) E_n + C(\Delta t)^{p+1},$$

where

$$\mathcal{L} := L \sum_{i=1}^{k} |\beta_i|.$$

Thus,

$$E_{n+1} \le (1 + \Delta t \mathcal{L}) E_n + C(\Delta t)^{p+1}.$$

Now a familiar argument can be used to prove that the error on the entire interval is $\mathcal{O}((\Delta t)^p)$.

Similar arguments prove convergence of AM methods and AB-AM PECE predictor-corrector methods.

Zero stability

Convergence is often proven by showing that the effects of small perturbations to a numerical solution are not amplified by more than a factor of $\mathcal{O}((\Delta t)^{-1})$.

Because the number of steps taken is also $\mathcal{O}((\Delta t)^{-1})$, we can think of this as saying that errors do no worse than add up.

A numerical method that has this property is said to be *zero-stable*.

The exact solution satisfies the numerical scheme plus a small perturbation ($\mathbf{lte}_n$).

Zero-stability then guarantees the error is $\mathcal{O}((\Delta t)^p)$; i.e., the method is convergent and of order p.

Zero-stability limits the achievable order of a multi-step method as a function of the number of past values used; e.g., BDFs of order higher than 7 are not zero-stable. In fact BDF6 is so close to not being zero-stable that it is not reliable in practice (hence the code `ode15s`).

When Δt varies, we should expect some restrictions on its rate of change to guarantee stability and convergence as $\max \Delta t \to 0$.

If the ratio of successive steps is uniformly bounded above, then stability and convergence can be proven for Adams methods [Shampine 1994].

This assumption holds in practice.

For BDFs the situation is slightly more complicated.

In practice, BDFs keep a fixed Δt until it appears advantageous to change to a different (constant) Δt.

Theory guarantees stability and convergence in such implementations provided the changes in Δt are limited in size and frequency, but these limits are not respected in practice (yet the codes work)!

Stability

The numerical methods used in practice are all stable as $\max \Delta t \to 0$, but what about for a finite Δt?

One can write down how small perturbations propagate in a numerical method, but the result is usually too complicated to lend any insight.

We consider instead standard stability analysis of the ODE itself.

Near a point $(t^*, \mathbf{y}^*)$, we approximate $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ by a linear, constant-coefficient equation

$$\dot{\mathbf{u}} = \mathbf{f}(\mathbf{y}^*) + \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}^*)(\mathbf{u} - \mathbf{y}^*).$$

(This process is known as *linearizing* the ODE about the (fixed) point $(t^*, \mathbf{y}^*)$.)

The difference between any two solutions of this (linear) equation satisfies the homogeneous *variational equation*

$$\dot{\mathbf{v}} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}^*)\mathbf{v}.$$

Usually we assume that the local Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}^*)$ is *diagonalizable*; i.e., there is a matrix $\mathbf{T}$ of eigenvectors such that

$$\mathbf{T}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}^*)\mathbf{T} = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_m).$$

If we now change variables to $\mathbf{w} := \mathbf{T}^{-1}\mathbf{v}$, we find that the equations *decouple*; i.e., each component of $\mathbf{w}$ satisfies $\dot{w}_i = \lambda_i w_i$.

This leads us to analyze the stability of numerical methods on the (scalar) *test equation*

$$\dot{y} = \lambda y,$$

where λ retains its interpretation as an eigenvalue of the local Jacobian.

Now if $\mathbb{R}(\lambda_i) > 0$ for some i, then w_j increases exponentially, and hence so does $\mathbf{v}$.

Because $\mathbf{v}$ represents the difference between two solutions of the linearized ODE, this means the ODE is (locally) unstable.

Conversely, if $\mathbb{R}(\lambda_i) \leq 0$ for all i, then w_i is bounded, and hence so is $\mathbf{v}$; the ODE is then (locally) stable.

With the test equation, we are able to analyze the stability of numerical methods by asking: if $\mathbb{R}(\lambda_i) \leq 0$ for all i so that the ODE is stable, how small must Δt be in order for the numerical method to also be stable?

This is known as *absolute stability theory*.

Despite the various approximations used, it has proven quite helpful in understanding how numerical methods behave in practice.

It can certainly be thought of as a necessary condition: any numerical method that does not perform well on the test equation will be of limited use for general problems.

As an example, consider Euler's method applied to the test equation:

$$y_{n+1} = y_n + \Delta t\, \lambda y_n = (1 + \Delta t\, \lambda) y_n.$$

For a bounded solution, we require $|1 + \Delta t\, \lambda| \leq 1$.

The set

$$S = \{|1 + z| \leq 1,\ \mathbb{R}(z) \leq 0\}$$

is called the *region of absolute stability* of Euler's method.

If $\Delta t\, \lambda \in S$, the method is stable, just like the ODE.

If $\Delta t\, \lambda \notin S$, the numerical solution will blow up!

More generally, we require all $\Delta t\, \lambda_j \in S$ for all eigenvalues λ_j of the local Jacobian.

Forward Euler will be stable for all λ *provided* Δt *is sufficiently small*; we knew this.[11]

Note 5. *For the test equation,* $L = |\lambda|$.

Classically, $L(b-0)$ *is not large; hence* Δt *does not have to be too small to preserve stability.*

But if $\mathbb{R}(\lambda) < 0$ *and* $|\lambda|(b-0) \gg 1$, *then the ODE is (very) stable, yet* $\Delta t \sim 1/|\lambda|$ *to maintain stability.*

The choice of Δt is determined by two criteria: accuracy and stability.

We would like it to be determined by accuracy; for stiff problems, it is determined by stability.

[11] This is what convergence means, and we know Forward Euler is convergent.

Consider the IVP

$$\dot{y} = -100y + 10, \qquad y(0) = 1,\ 0 \leq t \leq 10.$$

The exact solution is

$$y(t) = \frac{1}{10} + \frac{9}{10}e^{-100t}.$$

We notice there is an initial period of very rapid change[12].

In this region Δt must be small to approximate the solution accurately; usually this is small enough to maintain stability.

The problem is not stiff in the transient phase!

However, it does not take long for $y(t)$ to become approximately constant.

We should be able to take a large Δt in this case and still get a very accurate answer.

[12] This is called a *boundary layer* or *transient phase*.

However, for stability of FE we must have

$$|1 + \Delta t\,(-100)| \leq 1, \text{ or } \Delta t \leq 0.02;$$

i.e., if we try $\Delta t > 0.02$, the computation blows up!

Hence Δt is being chosen on the basis of stability, not accuracy.

Modern codes with step size control do not (usually) blow up.

They do try to increase Δt based on accuracy, so in fact they will take a number of unstable steps!

However, at some point the error becomes too large relative to the tolerance, resulting in step rejections and a reduction in Δt.

The typical result is an accurate solution, but not an efficient one due to the small step sizes and the many failed steps.

See `mildlyStiffDemo.m`

Stiff methods and A-stability

Consider the solution to the test equation with the backward Euler method:

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

This is a linear equation that we can solve for y_{n+1}:

$$y_{n+1} = \frac{1}{1 - \Delta t\, \lambda} y_n.$$

The stability region is thus

$$S = \left\{ \left| \frac{1}{1-z} \right| \leq 1,\ \mathbb{R}(z) \leq 0 \right\}.$$

Notice that this is in fact the whole left half of the complex plane.

Methods that have this property are called *A-stable*.

There is no step size restriction[13] due to stability.

In practice, backward Euler (or BDF1) has excellent stability properties.

BDF2 turns out to also be A-stable.

BDFs 3 through 6 have stability regions that extend to infinity, but they are restricted to sectors of the form

$$\{z = re^{i\theta} : r > 0,\ \pi - \alpha < \theta < \pi + \alpha\}$$

that contain the entire negative real axis.

This weaker version of A-stability is called A(α)-*stability*.

As the order increases, α becomes smaller, so that BDF6 is not robust in practice, and BDF7 is not stable at all.

All ERKs have stability regions that are finite; there exists a nice closed-form expression for them in terms of the coefficients of the Butcher tableau.

[13] at least not for the test equation!

So in particular, Heun's method (being an ERK) has a finite stability region.

Note that Heun's method can be interpreted as a forward-Euler predictor with a trapezoidal rule corrector.

However, trapezoidal rule turns out to be A-stable!

Thus, the stability properties of a predictor-corrector pair can differ greatly from the stability properties of the corrector formula!

For stability, we require that perturbations of the numerical solution do not grow.

But if $\mathbb{R}(\lambda) < 0$, then perturbations of the ODE decay (exponentially); it would be nice if the numerical method behaved similarly.

BDF1 (backward Euler) damps perturbations strongly when $\Delta t\, \mathbb{R}(\lambda) \ll -1$; specifically,

$$\lim_{\Delta t\, \mathbb{R}(\lambda) \to -\infty} y_{n+1} = 0.$$

An A(α)-stable method with this property is called L(α)-stable.

For trapezoidal rule, perturbations are barely damped in this limit; it is A-stable, but not L(α)-stable.

It is interesting to note that BDFs are stable for *all* $|\Delta t\,\lambda|$ sufficiently large; i.e., even when $\mathbb{R}(\lambda) > 0$, the numerical solution will decay (even though the exact solution does not)!

Because the method is convergent, everything is OK for $\Delta t\,\lambda$ sufficiently small; but if Δt is too large, then the heavy damping of these formulas may be undesirable.

Solving the nonlinear equations

We have seen that functional iteration is *not* a good way to solve the nonlinear equations at each step from an implicit method if the IVP is stiff: for the iteration to converge, Δt has to be small enough to make the mapping contractive.

This defeats the purpose of using an implicit method in the first place because we would like to take large Δt for stiff problems.

To solve these equations when the IVP is stiff, we must resort to a more powerful method: Newton's method (or variant thereof).

For BDFs, the nonlinear equations to be solved at each step have the form

$$\mathbf{y}_{n+1} = \Delta t\, \gamma \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}) + \boldsymbol{\psi},$$

where γ is a method-dependent constant, and $\boldsymbol{\psi}$ takes care of all the terms involving the past solution values $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, etc.

For example, for BDF1, $\gamma = 1$ and $\boldsymbol{\psi} = \mathbf{y}_n$.

BDF2 is given by

$$\mathbf{y}_{n+1} - \frac{4}{3}\mathbf{y}_n + \frac{1}{3}\mathbf{y}_{n-1} = \frac{2}{3}\Delta t \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1});$$

hence

$$\gamma = \frac{2}{3}, \qquad \boldsymbol{\psi} = \frac{4}{3}\mathbf{y}_n - \frac{1}{3}\mathbf{y}_{n-1}.$$

As with simple iteration for non-stiff problems, it is important to have a good initial guess $\mathbf{y}_{n+1}^{[0]}$.

A good approach is to fit a polynomial $\mathbf{P}(t)$ through $\mathbf{y}_n$, $\mathbf{y}_{n-1}$, . . . , $\mathbf{y}_{n-k+1}$ and set $\mathbf{y}_{n+1}^{[0]} = \mathbf{P}(t_{n+1})$.

To solve the nonlinear equations by Newton's method, we linearize about $(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]})$ to obtain

$$\mathbf{y}_{n+1}^{[\nu+1]} = \boldsymbol{\psi} + \Delta t\, \gamma[\mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]}) + \mathbf{J}(\mathbf{y}_{n+1}^{[\nu+1]} - \mathbf{y}_{n+1}^{[\nu]})],$$

where (for a true Newton iteration)

$$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]}).$$

It turns out that there are more efficient possibilities.

Similar methods, known as *variants* of Newton's method, turn out to be more efficient.

The idea here is that although you lose the quadratic convergence rate of Newton's method (hence requiring more iterations for a given accuracy), the iterations are so much cheaper that you can afford to take a few more and still end up with a favourable tradeoff.

Because the most expensive part of the true Newton iteration is evaluating and factoring $\mathbf{J}$, variants of Newton's method approximate $\mathbf{J}$ in different ways.

For example, the so-called *chord method* approximates $\mathbf{J}$ by it value at the beginning of the current step:

$$\mathbf{J} \approx \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_n, \mathbf{y}_{n+1}^{[0]}).$$

This leads to only one $\mathbf{LU}$ decomposition per step.

The method is sometimes said to *freeze* the Jacobian.

In practice, codes freeze Jacobians over as many steps as they can get away with.

It is important to organize the computation of Newton's method properly: the next iterate should be formed by computing a correction $\boldsymbol{\delta}\mathbf{y}^{[\nu]}$ to the current iterate as follows.

The Newton iteration can be re-written as

$$(\mathbf{I} - \Delta t\,\gamma\mathbf{J})\boldsymbol{\delta}\mathbf{y}^{[\nu]} = \boldsymbol{\psi} + \Delta t\,\gamma\mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}^{[\nu]}) - \mathbf{y}_{n+1}^{[\nu]}.$$

This is a linear system that can be solved for $\boldsymbol{\delta}\mathbf{y}^{[\nu]}$, and then the new iterate formed as

$$\mathbf{y}_{n+1}^{[\nu+1]} = \mathbf{y}_{n+1}^{[\nu]} + \boldsymbol{\delta}\mathbf{y}^{[\nu]}.$$

It is important to do this because when the IVP is stiff, the iteration matrix $(\mathbf{I} - \Delta t\,\gamma\mathbf{J})$ is *ill-conditioned*; i.e., is it close to singular.

When solving a linear system with an ill-conditioned coefficient matrix, you often get no more than a couple of digits of accuracy out of the solution.

This is all we would get out of $\mathbf{y}_{n+1}^{[\nu+1]}$ if we computed it directly; this is probably insufficient.

However, if we can compute a few correct digits in $\boldsymbol{\delta}\mathbf{y}^{[\nu]}$, then more and more digits would be correct in $\mathbf{y}_{n+1}^{[\nu+1]}$ as ν increases.

This is because $\boldsymbol{\delta}\mathbf{y}^{[\nu]}$ should be getting smaller with increasing ν: the right-hand side of the Newton iteration is the residual, and it should be getting smaller and smaller as our iterates get more and more accurate.

It is very convenient to have an option for codes to approximate Jacobians internally by finite differences.

This process can be thought of as a method for performing *numerical differentiation*.

Approximation of $\mathbf{J}$ by finite differences is the default option for the MATLAB solvers.

It is generally very difficult to find the right increment over which to difference when finding derivatives in this way; fortunately in this context all we need is a sufficiently accurate approximation for the modified Newton iteration to converge.

MATLAB has a built-in function called `numjac` that can generally be used to compute finite-difference Jacobians: it is not recommended to write your own code to do this!

In general, finite-difference Jacobians are convenient and often satisfactory; however, solvers are more robust (and often faster) if you can (correctly) compute and code an analytical Jacobian.

Nowadays, the use of *automatic differentiation software* allows you to have both the convenience of not having to figure out and code analytical Jacobians by hand as well as having the robustness as if you did.

This is an even more important issue in the numerical solution of BVPs.

For large systems of ODEs, especially those arising from a *method-of-lines* approximation of a system of PDEs, it is typical that only a few components of $\mathbf{y}$ appear in each equation.

If y_j does not appear in f_i, then $\frac{\partial f_i}{\partial y_j} = 0$.

When most of the entries of a matrix are 0, the matrix is said to be *sparse*.

By storing only the non-zero entries of a matrix, not only is storage greatly reduced, but so is the cost of solving a linear system involving it.

It is easy to take advantage of sparsity in the important case that a matrix is *banded*; i.e., all the non-zero elements line within a band of diagonals.

For example, if $J_{i,j} =$ for all (i, j) such that $|i-j| \leq 1$, then the matrix is *tri-diagonal* and has *bandwidth* of 3.

Such a system only requires $\mathcal{O}(m)$ storage instead of $\mathcal{O}(m^2)$ in general; it also requires only $\mathcal{O}(m)$ operations to factor, instead of $\mathcal{O}(m^3)$ in general.

These savings are critical in the solution of large systems; they often make the difference between obtaining a result or not.

More about this in Section 2.3.3.

As we have said, each step of a BDF method is much more expensive (both in terms of computing time and storage) than the corresponding step of a method for a non-stiff problem.

So the use of BDF methods are only advantageous when the increase in step size more than offsets the additional cost per step; i.e., for stiff problems.

For non-stiff problems, Adams methods are more accurate than BDFs and hence would be preferred.

It is hard to recognize whether a problem is stiff or non-stiff *a priori*; that's a shame because we know good methods for solving each kind and performance may suffer if you use the wrong one.

Unless you have previous insight as to the stiffness of a problem, the rule of thumb is to attempt to solve a problem using a non-stiff method and only revert to a stiff method if the performance is unsatisfactory.

A practical definition of a stiff problem is one that can be solved most efficiently by a method intended for stiff problems.

Stiff problems are typified by solution components that change on a scale that is small relative to the length of the interval of integration.

But the solution of interest itself must be slowly varying as well in some regions.

Many problems are mistakenly described as stiff because they have regions of rapid solution variation; e.g., boundary layers.

However, in these regions the problems are in fact non-stiff because the step size must be small to resolve these changes.

A problem is stiff in regions where its solution is *easy* to approximate (slowly varying), but large step sizes are precluded by stability considerations.

It is clear that the ability to change step sizes is crucial for the practical solution of these problems; see the proton transfer and Robertson problems of Section 1.4.

This also invites the possibility of software that detects stiffness and *switches methods automatically* between stiff and non-stiff solvers as appropriate; these codes are known as *type-insensitive software*.

Error Estimation and Change of Order

Estimating the local truncation error of a LMM is generally considered to be easier than estimating the local error of a RK method.

Even then, proving that these estimates work is hard!

With Adams methods, it is easy to take a step with formulas with orders that differ by 1; e.g., predict with AB$(k-1)$ and correct once with AMk.

As usual, advancing with the higher-order method AMk means we are doing local extrapolation.

This is the approach adopted in `ode113`.

Milne's device

Another approach is based on the expression

$$\mathbf{lte}_n = C_{p+1}(\Delta t)^{p+1}\mathbf{y}^{(p+1)}(t_n) + \ldots.$$

The idea here is to take a step with ABk to obtain $\mathbf{y}_{n+1}^{[0]}$ and AMk to obtain $\mathbf{y}_{n+1}$ and estimate $\mathbf{lte}_n$ by a suitable difference:

$$\mathbf{y}_{n+1} - \mathbf{y}_{n+1}^{[0]} = (C_{p+1} - \hat{C}_{p+1})(\Delta t)^{p+1}\mathbf{y}^{(p+1)}(t_n) + \ldots.$$

Hence we can obtain the computable estimate

$$\mathbf{lte}_n = \frac{C_{p+1}}{C_{p+1} - \hat{C}_{p+1}}(\mathbf{y}_{n+1} - \mathbf{y}_{n+1}^{[0]}).$$

This is known as *Milne's device*.

AMk is used to advance the integration because it is more accurate and stable than ABk.

The error of AMk is being controlled, so local extrapolation is not done.

Strictly speaking here we assume that the AM corrector is to be iterated to convergence, but because the order of the predictor-corrector combination is the same as the AM method, this approach applies to predictor-corrector combinations as well.

A natural approach for BDF methods

Recall that for BDF1,

$$\mathbf{lte}_n = -\frac{(\Delta t)^2}{2}\ddot{\mathbf{y}}(t_n) + \dots.$$

A natural approach for BDF methods is to estimate this quantity directly.

BDFs are based on numerical differentiation, so it is natural to interpolate $\mathbf{y}_{n+1}$, $\mathbf{y}_n$, and $\mathbf{y}_{n-1}$ with a quadratic polynomial $\mathbf{Q}(t)$ and set $\ddot{\mathbf{y}}(t_n) \approx \ddot{\mathbf{Q}}(t_n)$.

This is how `ode15s` estimates $\mathbf{lte}_n$.

Similarly a cubic interpolant is used in `ode23t` to approximate the third derivative term in $\mathbf{lte}_n$ of the trapezoidal method.

Both of these examples are one-step methods but they use past information to estimate $\mathbf{lte}_n$.

Past information is also used to construct initial guesses to implicit methods.

In the sense, the distinction between implicit one-step methods and multi-step methods is blurred.

Changing Order

An interesting aspect of these estimates is that they can be used to estimate the value of $\mathbf{lte}_n$ that would have been obtained if a different order had been used.

We can see this by looking at AM1 and AM2.

When taking a step with AM2, we could just as easily estimate $\mathbf{lte}_n$ for AM1.

It is also clear that (at least theoretically) we could use more past values to estimate the error that would have been made with AM3.

This opens the door to adapting the order as well in order to use larger step sizes.

This is what modern Adams and BDF codes like `ode113` and `ode15s` actually do.

Variation of order plays another role in modern multi-step codes.

Because the lowest-order formulas reduce to one-step methods, we can use them to start the integration; this is both convenient and efficient, and hence all modern LMM codes do this.

The past values from successful steps can then be used with formulas of higher order.

Again, we omit many of the practical details, but the idea is that order is quickly increased to a level suitable for the problem.

Codes that vary order and time-step are generally called variable-step size, variable-order (VSVO) codes.

They are not limited to LMM formulas; i.e., VSVO-RK codes exist as well.

As a final remark, we note that little is known theoretically about the behaviour of methods that vary order (i.e., formula).

There are results for constant order that provide insight, but in general our theoretical understanding is limited for modern variable-order LMM codes.

Continuous Extensions

Adams and BDF methods are based on polynomial interpolation; hence it is natural to use these polynomials as interpolants to provide continuous extensions for these methods.

These continuous extensions can help us change the step size in a linear multi-step method.

We have seen why there are practical reasons to work with a fixed Δt for some number of steps.

In order to use a LMM with a different "constant" step size ΔT starting at time t_n, we simply evaluate the continuous extension at $t_n - \Delta T$, $t_n - 2\Delta T$, etc.

This is a standard technique for changing step size with BDF methods; some codes (e.g., DIFSUB) also use it for Adams methods.

2.3 Solving IVPs in MATLAB

The simplest instance of solving an IVP in MATLAB is to just define the IVP and choose a solver.

Defining the IVP involves defining a function to evaluate the right-hand side $\mathbf{f}(t, \mathbf{y})$, an interval of integration $[t_0, t_f]$, and an initial condition $\mathbf{y}_0$.

We have said that the rule of thumb is to start with a non-stiff solver (like `ode45`) and move on to a stiff solver (like `ode15s`) if you suspect the problem may be stiff; e.g., if `ode45` performs unsatisfactorily.

However, a stiff solver will only perform well if the reason the non-stiff solver performed poorly was due to stiffness.

There are other things that make IVPs hard; if so, a stiff solver may not perform any better.

Note 6. MATLAB *programs can be written as* scripts *or as* functions*. The codes in the text are written as functions because then any auxiliary functions can be included in the same file as the main program.*

This becomes an issue when several such functions must be supplied; e.g., when solving ODEs with event functions.

It is always necessary to supply auxiliary functions when solving BVPs.

Example 2.3.1

$$\dot{y} = y^2 + t^2, \qquad y(0) = 0,\ 0 \leq t \leq 1.$$

This problem can be solved analytically.

It is also straightforward to solve numerically.

See `ch2ex0.m`

Example 2.3.2

Recall the proton transfer problem from Section 1.4.

$$\begin{aligned}\dot{x}_1 &= -k_1 x_1 + k_2 y,\\ \dot{x}_2 &= -k_4 x_2 + k_3 y,\\ \dot{y} &= k_1 x_1 + k_4 x_2 - (k_2 + k_3) y,\end{aligned}$$

subject to the initial conditions

$$x_1(0) = 0,\ x_2(0) = 1, y(0) = 0,$$

for $0 \leq t \leq 8 \times 10^5$. The constants are

$$\begin{aligned} k_1 &= 8.4303270 \times 10^{-10}, \quad k_2 = 2.9002673 \times 10^{11},\\ k_3 &= 2.4603642 \times 10^{10}, \quad k_4 = 8.7600580 \times 10^{-6}.\end{aligned}$$

See `ch2ex1.m`

Note 7. *Options are passed to the* MATLAB *IVP solvers via the auxiliary function* `odeset` *and the use of* keywords*.*

The command `help odeset` *gives short descriptions of the options; simply entering* `odeset` *gives you a brief reminder.*

Options can be assigned in any order.

Keywords are not case-sensitive.

Note 8. *This problem is stiff!*

Example 2.3.3

By default, the IVP solvers return the solution at all steps taken during the integration.

To get finer control over the output, you may specify an *output function* that the solver will call after each step with the solution it has just computed.

There are a few built-in output functions, e.g., `odeplot`, `odephas2`, `odephas3`, and `odeprint`.

By default, when you do not specify any output function, the solver assumes it is `odeplot`.

This function plots all the solution components as they are computed.

Usually you will want finer control than this: to select which components are displayed, you can use the `OutputSel` option.

Recall the program `dfs.m` that provides a modest capability for computing and plotting solutions of the scalar ODE $\dot{y} = f(t, y)$.

We will now see how `dfs.m` allows the user to specify a minimum allowable step size for an integrator.

A minimum allowable step size is a feature of all (high-quality) numerical IVP solvers; some allow the user to control it, others (including MATLAB) do not.

See `dfs.m`

2.3.1 Event Location

The solutions produced by `dfs.m` are plotted in a window specified by the array $[wL, wR, wB, wT]$.

The solution $y(t)$ starts at an initial point (t_0, y_0) and is plotted as long as $x \in [wL, wR]$ and $y \in [wB, wT]$.

A call is made to `ode45` with interval $[t_0, wR]$, but it entirely possible that the solution exit the window out the top or bottom before ever reaching wR.

We would like a way to terminate the integration if there is a t^* such that $y(t^*) = wT$ or $y(t^*) = wB$.

These are known as *events*; in general they are functions of the form

$$g_1(t, \mathbf{y}(t)) = 0,\ g_2(t, \mathbf{y}(t)) = 0, \ldots, g_k(t, \mathbf{y}(t)) = 0.$$

The process of determining the first time t^* such that one of these equations holds is called *event location*.

Sometimes we just want to know the time t^* or the solution $y(t^*)$ at the event; other times we may need to terminate the integration at t^* and/or start solving a new IVP (possibly with initial conditions that depend on t^* and/or $y(t^*)$).

Sometimes it matters whether the event function is increasing or decreasing at the time of the event.

In the program `dfs.m`, there are 2 event functions ($wB - y$ and $y - wT$) to detect solutions that leave the visible window.

In this case, both events are terminal, and whether the event function is increasing or not at the event is irrelevant.

All the MATLAB solvers have event location capability.

This is a non-trivial feature, and this fact is not always appreciated!

Event location problems can be ill-posed!

Most codes that locate events look for a change in sign in the components of $\mathbf{g}(t, \mathbf{y})$ at each step.

If $g_i(t_n, \mathbf{y}_n)$ and $g_i(t_{n+1}, \mathbf{y}_{n+1})$ have opposite signs, then the solver invokes a standard root finder to solve the algebraic equation $g_i(t, \mathbf{y}(t)) = 0$ for $t \in [t_n, t_{n+1}]$.

This is where a continuous extension is indispensable: without it we would have to take a step with the IVP solver for every t where we needed to evaluate $g_i(t)$.

It is much more efficient to just evaluate the continuous extension $\mathbf{S}(t)$; recall $\mathbf{S}(t) \approx \mathbf{y}(t)$ for all $t \in [t_n, t_{n+1}]$.

Event location then corresponds to finding the first (earliest) root of $\mathbf{g}(t, \mathbf{S}(t)) = \mathbf{0}$.

Strictly speaking we must find the root closest to t_n because, e.g., the ODE may change then.

Now $\mathbf{g}(t, \mathbf{y})$ has more than one component, so they must be processed simultaneously: it is entirely possible that while trying to find the root of $g_i(t, \mathbf{y})$ we find that $g_j(t, \mathbf{y})$ vanishes first!

The approach to event location just outlined has the usual pitfalls of completely missing even-order zeros (because there is no change of sign) and completely missing multiple zeros within one step.

Step sizes are chosen to resolve changes in $\mathbf{y}(t)$, *not* in $\mathbf{g}(t, \mathbf{y}(t))$: a code could easily step over multiple events and not detect a sign change.

Another issue is that we want the *earliest* root of $\mathbf{g}(t, \mathbf{y})$, and in general there is no way to guarantee we have computed it.

Discontinuous event functions are not rare, and this complicates the event location process further.

The root-finding problem can be ill-conditioned; i.e., t^* is poorly determined by evaluating $\mathbf{g}$.

Yet another complication here is that in the pure root-finding problem, we assume that $\mathbf{g}$ can be evaluated very accurately (to within a multiple of machine precision); with event location $\mathbf{y}(t)$ is known to much less accuracy.

This leads to the question of how accurately one should determine t^*.

It is difficult to choose tolerance values for event location in addition to those for solving the IVP.

In MATLAB, solvers locate events as accurately as possible (down to machine precision).

For this to be practical, the root-finding algorithm needs to be fast in the usual case when $\mathbf{g}$ is smooth and reasonably fast when it is not.

The MATLAB solvers implement a hybrid strategy of the (relatively) slow but sure method of bisection with the so-called *Illinois algorithm* that is both fast for smooth $\mathbf{g}$ and vectorizable.

Event location boils down to specifying the events and the action of the solver when it finds one.

Event location problems usually get complicated by what needs to happen *after* the event is located.

See `dfs.m` for some simple examples of event location.

Example 2.3.4

Poincaré maps are important tools for studying the behaviour of dynamical systems.

They give snapshots of the state of the dynamical system at pre-defined instants.

However, rather than taking snapshots at fixed intervals of time, values are often sought for which a linear combination of solution components vanishes; e.g., when a given component is 0.

We consider a pair of coupled harmonic oscillators

$$\dot{\mathbf{y}} = (ay_3, by_4, -ay_1, -by_2)^T, \quad \mathbf{y}(0) = (5, 5, 5, 5)^T,$$

for $0 \leq t \leq 65$, where $a = 103/33$, $b = 19/9$.

We look at a plot of the part of phase space defined by $(y_1(t), y_2(t), y_3(t))$.

We also look at Poincaré maps defined by

$$(y_1(t^*), y_3(t^*)) \quad \text{when } y_2(t^*) = 0,$$
$$(y_1(t^*), y_2(t^*)) \quad \text{when } y_3(t^*) = 0.$$

See `ch2ex2.m`.

The output from the code is

```
Event 1 occurred 43 times.
Event 2 occurred 65 times.
```

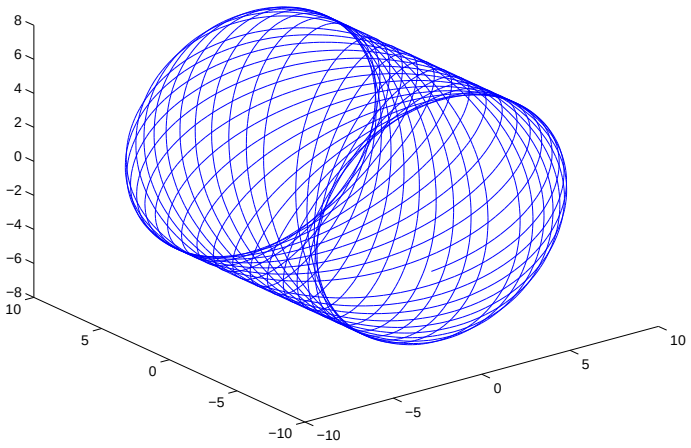


Figure 1: Part of phase space $(y_1(t), y_2(t), y_3(t))$.

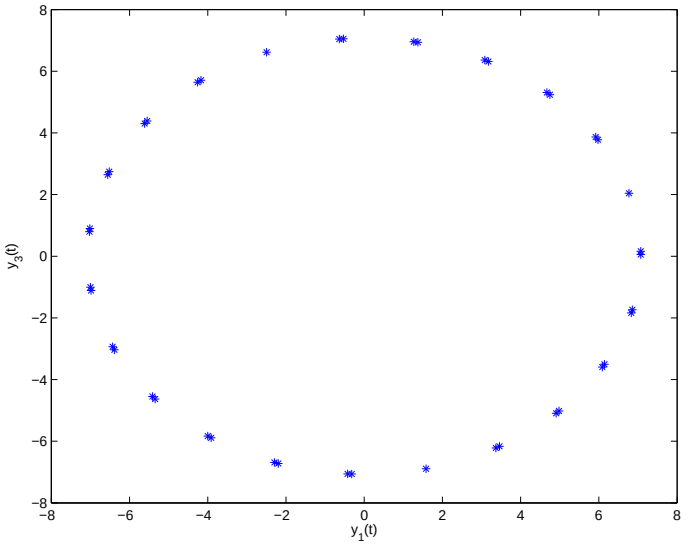


Figure 2: Poincaré map of $(y_1(t^*), y_3(t^*))$ when $y_2(t^*) = 0$.

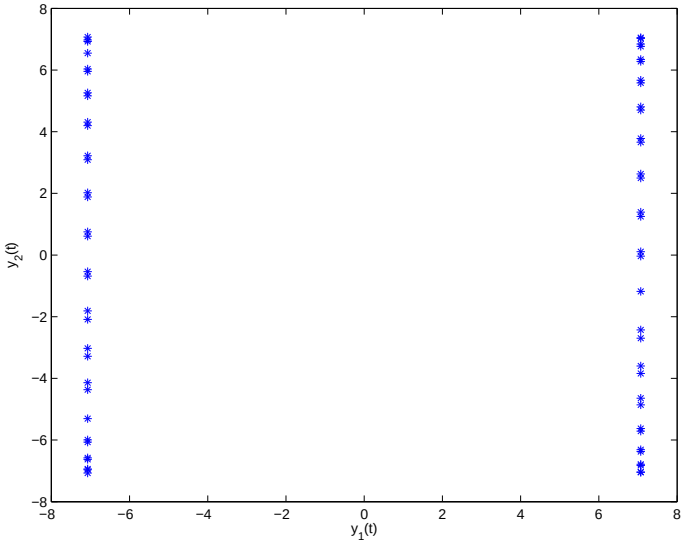


Figure 3: Poincaré map of $(y_1(t^*), y_2(t^*))$ when $y_3(t^*) = 0$.

Note 9. *Initial points are treated differently in event location because sometimes an event that we wish to regard as terminal occurs at the initial point; see, e.g., Example 2.3.5.*

Accordingly, solvers never allow initial points to trigger terminal events.

When using event location, the solver returns additional information, i.e., the times that the events occur (`te`), the solutions corresponding to these points (`ye`), and an integer array specifying which event function (component of $\mathbf{g}$) triggered the event (`ie`).

So a convenient way to check if there are no events is to test `isempty(ie)`.

Example 2.3.5

Consider the problem of a ball bouncing down a ramp.

Let the ramp be the long side of the triangle with vertices $(0,0)$, $(0,1)$, and $(1,0)$.

Let the position of the ball at time t be $(x(t), y(t))$.

Suppose that the ball is released from rest above the top of the ramp; i.e., $x(0) = 0$, $y(0) > 1$, $\dot{x}(0) = 0$, and $\dot{y}(0) = 0$.

The (free) motion of the ball is governed by the equations

$$\ddot{x} = 0, \quad \ddot{y} = -g,$$

where $g = 9.81$ is the acceleration due to gravity.

We convert this system to first order in the usual way by defining

$$y_1(t) = x(t),\ y_2(t) = \dot{x}(t),\ y_3(t) = y(t),\ y_4(t) = \dot{y}(t).$$

We need to detect when the ball hits the ramp because at that point we will need to modify the state of the ball to model the impact.

In this case, the equations of motion do not change, but they might in other problems.

Suppose the ball hits the ramp at $t = t^* > 0$.

Then the ball's horizontal position is $x(t^*)$ and its vertical position is $y(t^*) = 1 - x(t^*)$.[14]

Thus the (scalar) event function is

$$g_1 = y_3 - (1 - y_1).$$

This is terminal event because we want to change things about the simulation at that point (we do not want to just keep going).

We note that there is nothing in the ODEs themselves that tells the ball that the ramp is there!

[14] The first event will have $x(t^*) = 0$.

After hitting the ramp, the ODEs remain the same, but the initial conditions are reset as follows:

The position of the ball after collision is the same as it was before collision.

The velocity of the ball normal to the ramp before collision is reversed and reduced by a factor $k \in (0, 1)$ known as the *coefficient of restitution*; the velocity parallel to the ramp remains unchanged[15].

Without getting into the physics, after collision, we start a new IVP with the same ODEs and ICs given by

$$(y_1(t^*), -ky_4(t^*), y_3(t^*), -ky_2(t^*)).$$

The same terminal event function can be used to detect subsequent collisions.

However, the initial point of these new IVPs will all be touching the ramp: this is why we treat initial points differently for terminal events.

[15] The true physics are significantly more complicated!

We can avoid reporting the initial point of each sub-integration as an event by setting `direction` to -1: the beginning of each sub-integration is characterized by the ball leaving the ramp; i.e., g_1 is increasing.

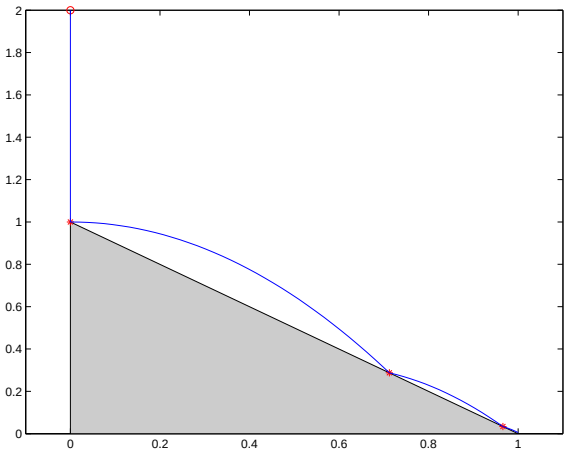
The end of each sub-integration is characterized by the ball hitting the ramp; i.e., g_1 is decreasing.

Setting `direction`$= -1$ indicates that only g_1 decreasing should trigger an event.

We also terminate the simulation when the ball leaves the ramp.

This corresponds to $x(t^*) = 1$; hence we set another event function $g_2 = y_1 - 1$.

See `ch2ex3.m`



For some choices of k and $y(0)$, it is possible the ball never reaches the end of the ramp.

Our model is not valid then (the ball is rolling), so we also terminate if bounces are clustering (successive t^*'s differ by $< 1\%$) or if a bounce occurs within 1% of the end of the ramp.

Note 10. *We do not know how long it will take for the ball to reach the end of the ramp (if ever!)!*

But solvers require a final time!

So we take a guess at one for the first bounce, then re-use this guess for subsequent bounces, but we also choose a short enough interval to give us a nice plot.

Note 11. *The path of the ball consists of several pieces that are accumulated in a couple of arrays for convenience.*

Because the ODEs are so simple, the solver output is so coarse that the plot of the solution would look jagged. This is taken into account in the choice of the intervals of integration.

2.3.2 ODEs having a Mass Matrix

The MATLAB solvers accept IVPs in the more general form

$$\mathbf{M}(t, \mathbf{y})\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \tag{3}$$

where $\mathbf{M}(t, \mathbf{y})$ is called a *mass matrix*.

If $\mathbf{M}(t, \mathbf{y})$ is singular, then this is a system of *differential-algebraic equations* (DAEs).

DAEs are not ODEs!

A treatment of the numerical solution of DAEs is well beyond the scope of this course; we note only that the MATLAB codes for stiff ODEs can solve so-called *index-1* DAEs.

In its simplest use, solving ODEs of the form (3) is done simply by telling the solver a mass matrix is involved via the option `Mass`.

If $\mathbf{M}(t, \mathbf{y})$ is constant, then you should pass this constant matrix itself as the value of the `Mass`.

The codes exploit the fact that $\mathbf{M}$ is constant, so it is efficient and convenient for the user.

The solvers actually solve the standard form

$$\dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y}) := \mathbf{M}^{-1}\mathbf{f}(t, \mathbf{y}),$$

where an $\mathbf{LU}$ factorization of $\mathbf{M}$ is computed before the integration process begins.

Whenever $\mathbf{F}(t, \mathbf{y})$ is to be evaluated, a linear system is solved using $\mathbf{f}(t, \mathbf{y})$ using the pre-computed factorization.

When $\mathbf{M}(t, \mathbf{y})$ is not constant, you must provide a (sub)function to evaluate it and pass the name to the solver as the value of `Mass`.

This is the only thing that is strictly required to solve a problem.

If you know that $\mathbf{M}(t_0, \mathbf{y}_0)$ is not singular, then you can help the solver by setting the option `MassSingular` to `no`.

Conversely, if you know you have a DAE, you can set `MassSingular` to `yes`.

If you do not know or you do not set `MassSingular`, the solver sets `MassSingular` to `maybe` and numerically checks for singularity itself.

Note that the codes only check for singularity of $\mathbf{M}(t_0, \mathbf{y}_0)$.

If $\mathbf{M}(t, \mathbf{y})$ becomes singular at a later time (or state), most codes will behave badly, except perhaps `ode15i`, which is designed to handle such a case more gracefully.

For large systems of ODEs, it is essential to take advantage of sparsity to avoid, e.g., storing 0's, multiplying by 0, or adding 0.

These are wasteful operations, and if $\mathbf{M}(t, \mathbf{y})$ is say more than 90% sparse, the overall computation will bog down even for moderately sized systems!

Also there is an option to inform the solver how strongly $\mathbf{M}(t, \mathbf{y})$ depends on $\mathbf{y}$, but it may soon be removed.

More on this after Example 2.3.11.

Example 2.3.6

There is a demo in MATLAB called `batonode` that illustrates the use of a mass matrix.

A baton is modelled as 2 particles of mass m_1 and m_2 that are attached to opposite ends of a light straight rod of length L.

The baton travels in a vertical plane under the action of gravity.

If the co-ordinates of the first particle at time t are $(x(t), y(t))$ and the rod makes angle $\theta(t)$ with the horizontal, then Lagrange's equations of motion are naturally expressed in terms of a mass matrix that depends on $\theta(t)$.

See `batonode.m`

It is certainly more convenient to solve the ODEs in this form rather than to find the inverse of $\mathbf{M}$ analytically to convert it to standard form.

With such a small system, the numerical test to see if $\mathbf{M}$ is singular with the default setting of `MassSingular` to `maybe` is not very expensive.

Not much efficiency is gained by taking advantage of the sparsity of $\mathbf{M}$ in this case either because of the small system size.

In general to take advantage of sparsity, only the line `M=zeros(6,6)` need be changed to `M=sparse(6,6)`.

Example 2.3.7

The incompressible Navier–Stokes equations for time-dependent fluid flow in one space dimension on an interval of length L can be formulated as the system of PDEs

$$\frac{\partial \mathbf{U}}{\partial t} + \mathbf{A}\frac{\partial \mathbf{U}}{\partial z} = \mathbf{C}, \quad 0 \leq z \leq L,\ t \geq 0,$$

where $\mathbf{U} = (\rho, G, T)^T$ consists of 3 unknowns, the density ρ, the flow rate G, and the temperature T,

$$\mathbf{C} = \begin{pmatrix} 0 \\ -KG\left|\frac{G}{\rho}\right| - \rho g_a \sin(\theta) \\ \frac{a^2 \Phi P_H \kappa}{C_p A_f} \end{pmatrix},$$

and

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{1}{\rho\kappa} - \frac{G^2}{\rho^2} & 2\frac{G}{\rho} & \frac{\beta}{\kappa} \\ -\frac{a^2\beta\bar{T}G}{\rho^2 C_p} & \frac{a^2\beta\bar{T}}{\rho C_p} & \frac{G}{\rho} \end{pmatrix}.$$

See

`ch2ex4.m`

for relevant fluid properties and parameter values.

We note that $\bar{T} = T + 273.15$ and the boundary conditions are

$$\rho(0,t) = \rho_0 = 795.5,$$
$$T(0,t) = T_0 = 255.0,$$
$$G(L,t) = G_0 = 270.9.$$

We wish to compute a *steady-state* solution to these equations; i.e., for $\mathbf{U}_t = 0$.

It turns out that if $\rho_t = 0$ then $G(z) = G_0$ for all $z \in [0, L]$ so the PDEs reduce to the ODEs

$$\begin{pmatrix} \frac{1}{\rho\kappa} - \frac{G_0^2}{\rho^2} & \frac{\beta}{\kappa} \\ -\frac{a^2\beta\bar{T}G_0}{\rho^2 C_p} & \frac{G_0}{\rho} \end{pmatrix} \begin{pmatrix} \frac{d\rho}{dz} \\ \frac{dT}{dz} \end{pmatrix} = \begin{pmatrix} -KG_0 \left|\frac{G_0}{\rho}\right| - \rho g_a \sin(\theta) \\ \frac{a^2\Phi P_H \kappa}{C_p A_f} \end{pmatrix}.$$

This way of converting PDEs to ODEs is known as *continuous space, discrete time* (CSDT) solution.

It can also be called the *transverse method of lines*.

Such an approach may lead to an IVP or a BVP, depending on the BCs for the PDE.

These equations are used to model the sub-cooled liquid portion of a three-phase[16] steam generator in power systems.

In such a model, the (moving) boundaries between the phases are determined using the equation of state.

[16] The other two phases are the saturated liquid-steam phase and the pure steam phase.

For example, the ODEs can be integrated from $z = 0$ in the positive z direction until $\rho = \rho_{\text{sat}}(T)$, the "liquid-side" saturation density.

We can locate this point by defining the event function

$$g(z, \rho, T) = \rho(z) - \rho_{\text{sat}}(T(z)).$$

For convenience, we use the phony equation of state

$$\rho_{\text{sat}}(T) = -3.3(T - 290) + 738.$$

See `ch2ex4.m`

2.3.3 Large Systems and the Method of Lines

There are issues related to solving large systems of ODEs that we now address directly.

MATLAB is a PSE and hence not necessarily appropriate for solving the very large systems that are commonplace in scientific computing; nonetheless it is feasible to solve rather large systems.

The *method of lines* (MOL) is a way of approximating PDEs by ODEs.

Often these systems are large and stiff.

MATLAB has a code called `pdepe` that solves small systems of elliptic and parabolic PDEs in one space dimension and time.

The idea behind the MOL is to discretize all the (spatial) variables to obtain a set of ODEs.

This process is called *semi-discretization*.

Example 2.3.8

We solve the one-way wave equation[17]

$$u_t + c(x)u_x = 0,\ c(x) = \frac{1}{5} + \sin^2(x-1),$$
$$0 \le x \le 2\pi,\ 0 \le t \le 8,\ u(x,0) = e^{-100(x-1)^2},$$

and *periodic* boundary conditions $u(0,t) = u(2\pi,t)$.

The initial profile is not periodic, but it decays so fast near the ends that it can be considered so.

The solutions to this equation are waves that move with speed $c(x)$.

In this case, we have a peak at $x = 1$ that moves right (because $c(x) > 0$) with a variable speed.

It turns out the peak does not reach the right end-point during the time interval specified.

[17] This is also an example of a one-dimensional *hyperbolic conservation law.*

We note that because information propagates from left to right in this problem, only a BC at $x = 0$ should be specified.

In the MOL, a grid $x_1 < x_2 < \ldots < x_m$ is chosen to partition the spatial domain $[0, 2\pi]$.

We define the unknowns to be the solution at the grid points: $u_i(t) \approx u(x_i, t)$ for $i = 1, 2, \ldots, m$.

These functions are determined by a system of ODEs

$$\frac{du_i}{dt} = -c(x_i)(Du)_i,\ u_i(0) = u(x_i, 0),\ i = 1, 2, \ldots, m,$$

where $(Du)_i \approx u_x(x_i, t)$.

We first describe how to approximate u_x using a *spectral method*: $u_x(x_i, t)$ is approximated by interpolating the values $u_1(t),\ u_2(t),\ \ldots, u_m(t)$ with a trigonometric polynomial in x, differentiating the interpolant with respect to x, and evaluating the derivative at x_i.

It is typical of spectral methods to use data from the entire interval in this way.

For equally spaced mesh points, this approximation is made very convenient using the *fast Fourier transform* (FFT); we omit details.

See `ch2ex5.m`

Note 12. *With large problems, you should not store the output from* every *step!*

Nor should you return the numerical solution as a structure (way too much information).

Note 13. *It is commonly thought that any moderate to large system of ODEs that arise from the MOL is stiff, but this is not necessarily true!*

If we run `ch2ex5.m` *with* `ode15s`*, we find it takes about 5 times longer than with* `ode23`*!*

Example 2.3.9

Spectral approximations are very accurate for smooth functions, but they are global in nature and not trivial to understand.

On the other hand, *finite difference approximations* are easier to understand and implement.

They are of (much) lower accuracy, but they are local in nature and hence require (much) less smoothness of the solution.

It is worth mentioning that hyperbolic conservation laws (like the one-way wave equation) support discontinuous solutions, and this is a matter that receives considerable attention in the numerical solution of such PDEs.

A simple, yet reasonable finite difference scheme for hyperbolic conservation laws is the so-called first-order *upwind* scheme.

Because $c(x) > 0$, the wave always moves to the right, and the upwind spatial discretization leads to

$$\dot{u}_i = -c(x_i)\frac{u_i - u_{i-1}}{\Delta x},\ i = 1, 2, \ldots, m,$$

where we have assumed a uniform mesh in x with equal spacing Δx.

The periodic boundary condition is imposed for $i = 1$,

$$\dot{u}_1 = -c(x_1)\frac{u_1 - u_m}{\Delta x}.$$

Rather than use this, the initial data allow us to specify $u_x(0, t) = 0$, which implies $\dot{u}_1 = 0$.

See `ch2ex6.m`

This IVP is not stiff, but it is illustrative to see what are some of the issues involved in treating it as a stiff problem.

Stiff solvers must form and factor Jacobian matrices; for a system of size m, the Jacobian is $m \times m$.

Values of m in the thousands or millions are common, making it critical to account for sparsity, not just for storage but also for efficiency in solving the linear systems.

Sparse matrix technology is fully integrated within MATLAB.

The only typical provision for sparsity in general software for scientific computing is for *banded Jacobians*.

Somehow we must inform the solver of the zeros in $\mathbf{J}$.

The simplest way is to provide $\mathbf{J}$ analytically as a sparse matrix.

This is done by means of the option `Jacobian`; if $\mathbf{J}$ is constant, the option is set equal to the constant value.

This could have been done in `ch2ex6.m` with

```
B = [ [ -c_h(2:N); 0] [0; c_h(2:N)] ];
J = spdiags(B,-1:0,N,N);
options = odeset('Jacobian',J);
```

When $\mathbf{J}$ is not constant, you provide a function to evaluate it and return it as a sparse matrix.

The solvers are generally somewhat more efficient and robust when using analytical Jacobians, so it is better than letting the solver work out Jacobians numerically.

For large problems it is important to inform the solvers of the Jacobian's sparsity structure.

This is provided by creating a matrix that has zeros in the places where $\mathbf{J}$ has zero entries and ones elsewhere; it is passed to the solver in the option `JPattern`.

Here is an example for `ch2ex6.m`

```
S = spdiags(ones(N,2),-1:0,N,N);
S(1,1) = 0;
```

For this problem it would be more efficient to provide an analytical expression for $\mathbf{J}$ (because it is not difficult to compute) and passed as the value of `Jacobian` (because it is constant).

Example 2.3.10

We have just seen that one way to speed up the numerical approximation of Jacobians is to use sparsity.

Another way is to make function evaluations more efficient.

One way to do this in MATLAB is via *vectorization*.

It is a form of parallel processing: independent processes are performed simultaneously rather than sequentially.

It is perhaps more precisely called *single instruction multiple data* (SIMD).

For example, when the solvers approximate a Jacobian, they evaluate $\mathbf{f}(t, \mathbf{y})$ for several $\mathbf{y}$ and the same t.

This can be exploited by coding the function to accept arguments `(t,[y1 y2 ...])` and return `[f(t,y1) f(t,y2) ...]`.

You tell the solver this is the case by setting the option `Vectorized` to `on`.

It is often not much work to vectorize a function in MATLAB.

For example, we can vectorize the right-hand side in `brussode`

```
dydt(i) = 1 + y(i+1)*y(i)^2 - 4*y(i) + ...
             c*(y(i-2) - 2*y(i) + y(i+2));
```

by re-writing this line as

```
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:) - 2*y(i,:) + y(i+2,:));
```

Whether vectorization is helpful or not depends on the sparsity pattern of $\mathbf{J}$ and how expensive it is to evaluate $\mathbf{f}$.

To illustrate a vectorization that is not as straightforward, suppose we wish to vectorize the right-hand side from `ch2ex5.m`

```
dvdt = mc.*real(ifft(ixindices.*fft(v)));
```

MATLAB's `fft` function is already vectorized, so the key will be to vectorize the operation `ixindices.*fft(v)`.

We could do it in a loop, but a more efficient way is to make `ixindices` a matrix with `N` identical columns.

```
ixindices = repmat(ixindices,1,N);
```

We can now use the `size` command to find out how many columns there are in `v` and use the corresponding number of columns in `ixindices` to do the multiplication `ixindices.*fft(v)`.

The same change must be made with `mc`.

Finally, the evaluation of `dvdt` is

```
nc = size(v,2);
dvdt = mc(:,1:nc).*real(ifft(ixindices(:,i:nc).*fft(v)));
```

Example 2.3.11

Mass matrices arise when a *Galerkin method* is used for the spatial discretization by the *finite element method*.

Consider solving the heat equation

$$u_t = u_{xx}, \quad 0 \leq x \leq 1, \ t \geq 0,$$

subject to the boundary conditions

$$u(0,t) = 0, \quad u(1,t) = 1,$$

and initial conditions

$$u(x,0) = x + \sin(\pi x).$$

An approximate solution is sought in the form

$$v(x,t) = \sum_{j=1}^{m} S_j(x) v_j(t).$$

For a given t, $v(x,t)$ satisfies the PDE with residual

$$R(x,t) = v_t(x,t) - v_{xx}(x,t).$$

Note that if $R(x,t) \equiv 0$, $v(x,t)$ would be the exact solution to the PDE.

The Galerkin method requires that $R(x,t)$ be orthogonal to the space spanned by the basis functions[18] $S_j(x)$:

$$(R, S_j) = (v_t, S_j) - (v_{xx}, S_j) = 0, \quad \text{for each } j, \quad (4)$$

where the inner product (f,g) of two functions $f,\ g \in C[0,1]$ is defined by

$$(f,g) := \int_0^1 f(x)g(x)\,dx.$$

[18] The $S_j(x)$ are also known as *shape functions*.

Suppose each $S_j(x)$ is a piecewise-linear function satisfying $S_j(x_i) = 1$ if $i = j$ and 0 otherwise[19]; then

$$v(x_j, t) = v_j(t) \approx u(x_j, t).$$

Substituting the form of $v(x, t)$ into (4) and doing an integration by parts yields

$$\sum_{j=1}^{m} (S_j, S_i)\dot{v}_j + \sum_{j=1}^{m} \left(\frac{dS_j}{dx}, \frac{dS_i}{dx} \right) v_j = 0.$$

These inner products can be worked out on an arbitrary mesh, but if we assume it is uniform with spacing Δx, we obtain

$$\frac{1}{6}\dot{v}_{j-1} + \frac{4}{6}\dot{v}_j + \frac{1}{6}\dot{v}_{j+1} = \frac{v_{j-1} - 2v_j + v_{j+1}}{(\Delta x)^2}.$$

[19] Such shape functions are known as "hat" functions.

We use the boundary conditions

$$v_0(t) = 0, \quad v_{m+1}(t) = 1,$$

in the first and last ODEs respectively.

See `ch2ex7.m`

Alternatively, one could augment the system of ODEs with

$$\dot{v}_0 = 0, \quad \dot{v}_{m+1} = 0,$$

which says that v_0 and v_{m+1} are constants, and the constant values are taken from the ICs.

Fully implicit ODEs

There are a few codes that solve fully implicit ODEs of the form

$$\mathbf{F}(t, \mathbf{y}, \dot{\mathbf{y}}) = \mathbf{0}. \tag{5}$$

One of the most popular is `DASSL`; MATLAB has one called `ode15i`.

This general form includes DAEs.

Solving DAEs can be much more difficult than ODEs in theory and practice.

An obvious theoretical difficulty is that an initial value $\mathbf{y}_0$ may not be enough to specify a solution: we must also find a value $\dot{\mathbf{y}}(t_0)$ that is consistent with (5).

It is also clear that the solvers will need ways to compute $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{y}}}$ as well as $\frac{\partial \mathbf{F}}{\partial \mathbf{y}}$.

This complicates the user interface and storage management.

As mentioned, for large systems it is important to account for sparsity in the Newton iteration matrix.

For BDF methods, the iteration matrix takes the form $\mathbf{M} - \Delta t \gamma \mathbf{J}$, so we have to account for the structures of both $\mathbf{M}$ and $\mathbf{J}$.

Because of these complications, many codes do not handle the presence of mass matrices directly.

MATLAB can do it in a straightforward way because it simply treats everything as a general sparse matrix.

However, the strategies for constructing and updating the iteration matrix depend on how strongly $\mathbf{M}(t, \mathbf{y})$ depends on $\mathbf{y}$.

Generally, the weaker the dependence, the better.

We have already seen how to deal with constant $\mathbf{M}$ by passing it directly to the solver.

The information on the dependence of $\mathbf{M}$ on $\mathbf{y}$ is communicated to the solver through the option `MStateDependence`.

This option may soon be deprecated, so we are careful not to say too much.

`MStateDependence` can take on 3 possible values: `none`, `weak`, and `strong`.

If $\mathbf{M} = \mathbf{M}(t)$ only, `MStateDependence` $=$ `none`.

The default setting is `MStateDependence` $=$ `weak`.

This allows the full suite of approximations (in particular, freezing various matrices) to be used.

This may lead to poor performance; if so it may be more appropriate to use `MStateDependence` $=$ `strong`.

However, now the computations are much more expensive, and the strategies employed resemble more those employed by `ode15i` used to solve (5).

So it may make more sense to use `ode15i` for such problems.

2.3.4 Singularities

The theory supporting what is done in software assumes some smoothness of the solution and its derivatives; when this is not satisfied at an isolated point, we must supplement the codes with some analytical results.

One common technique is to approximate the solution of interest near the singular point with a series or asymptotic expansion; elsewhere it can be approximated by standard software.

Note 14. *"Solution of interest" implies the solution at a singular point may not be unique!*

This is much more common with BVPs, so we defer more discussion until later.

Example 2.3.12

A classical analysis of the collapse of a spherical cavity in a liquid leads to the IVP

$$(\dot{y})^2 = \frac{2}{3}(y^{-3} - 1), \quad y(0) = 1.$$

The (non-dimensional) variables are time t and cavity radius y.

The integration is to terminate when $y = 0$.

In standard form, the ODE is

$$\dot{y} = -\sqrt{\frac{2}{3}(y^{-3} - 1)},$$

where we have chosen the negative square root because we wish to model cavity *collapse*.

There are two difficulties:

1. The equation does not satisfy a Lipschitz condition in a neighbourhood including $t = 0$, so the solution may not be unique.

   In fact, $y(t) \equiv 1$ is another solution!

2. At total collapse, $\dot{y} = -\infty$.

We approximate $y(t)$ for $t \in [0, d]$ by a Taylor series.

We then take $y(d)$ as the initial condition to solve an IVP on $t \geq d$.

This can be done with the symbolic capabilities of MATLAB.

We assume an approximation of the form

$$y(t) = 1 + at + bt^2 + \dots.$$

Then the code

```
syms y t a b res
y = 1 + a*t + b*t^2
res = taylor(diff(y)^2 - (3/2)*(1/y^3-1),3)
```

substitutes the expression for y into the ODE and computes the first 3 terms in the Taylor series for the residual.

The result is

```
res = a^2+(4*a*b+2*a)*t+(4*b^2+2*b-4*a^2)*t^2
```

To satisfy the ODE as well as possible, we choose a and b to make terms as many terms as we can vanish, starting at the lowest order.

This leads to

$$a = 0,\ b = 0, -1/2.$$

It seems that the singular IVP has two solutions that can be expanded as a Taylor series.

The choice $b = 0$ leads to $y(t) \equiv 1$, a solution in which we are not interested.

The other solution is

$$y(t) = 1 - \frac{1}{2}t^2 + \ldots,$$

which is decreasing for small t; this is the solution with the physical behaviour that we want.

It is possible to continue this analysis to find that

$$y(t) = 1 - \frac{1}{2}t^2 - \frac{1}{6}t^4 - \frac{19}{180}t^6 \ldots.$$

To estimate how many terms we need to keep, we can estimate the error by the size of the first term neglected; for $d = 0.1$, we find that the first 3 terms leads to a relative error of about 10^{-7}.

See `ch2ex8.m`

We also interchange the independent and dependent variables to deal with the infinite slope when $y = 0$.

This is a useful technique, but we must be careful that the process is valid.

We can use y as the independent variable away from $t = 0$ because the ODE tells us that $y(t)$ is strictly monotone[20].

So we can solve the IVP

$$\frac{dt}{dy} = -\sqrt{\frac{3y^3}{2(1-y^3)}}, \quad t(y_d) = d,\ y \in [y_d, 0].$$

The time of total collapse is the value of t when $y = 0$.

[20] In this case, $y(t)$ is monotone decreasing.