Chapter 6 - *Ordinary Differential Equations*

7.1 Solving Initial-Value Problems

In this chapter, we focus on the solution of initial-value problems in ordinary differential equations.

Ordinary differential equations (ODEs) get their name from the fact that we are solving equations that involve *unknown functions* $\mathbf{y}(t)$ *of one independent variable* t and their derivatives $d\mathbf{y}/dt$, which we denote by $\dot{\mathbf{y}}(t)$.

Notes:

1. $\mathbf{y}(t)$ is really a vector of size m; i.e.,

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{bmatrix}.$$

However, it does not usually hurt (especially when first starting out) to think of it as a scalar (as we are more accustomed to).

Differentiation in this sense simply means to apply the derivative operator to each component individually; i.e.,

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \vdots \\ \dot{y}_m(t) \end{bmatrix}.$$

2. Higher derivatives are denoted using more dots; e.g.,

$$\frac{d^2\mathbf{y}(t)}{dt^2} = \ddot{\mathbf{y}}(t), \quad \text{etc.}$$

To begin, we will try to approximate values of $\mathbf{y}(t)$ for certain discrete values of t, denoted t_n, $n = 1, 2, \ldots$.

We denote the corresponding approximations by $\mathbf{y}_n$; i.e.,

$$\mathbf{y}_n \approx \mathbf{y}(t_n).$$

It will be possible to construct approximations to $\mathbf{y}(t)$ for *any* t using the values $\mathbf{y}_1, \mathbf{y}_2, \ldots$. (how?)

Note: *Partial differential equations* (PDEs) involve functions of *more than one variable*; e.g., $u = u(x, t)$.

In this case, derivatives can be with respect to x or t.

Perhaps it is not hard to imagine that solving partial differential equations is much harder than solving ordinary differential equations!

However there is a common technique (called the *method of lines*) whereby a partial differential equation is converted to a set of ordinary differential equations for its numerical solution.

Most modern software packages are designed to solve explicit ODEs:

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

Of course, we appreciate that Just having an ODE is not enough to completely specify the problem.

This is because if $\mathbf{y}(t)$ solves the above ODE, then so will $\mathbf{y}(t) + C$ for any constant C.

We need an extra condition to uniquely determine $\mathbf{y}(t)$!

This can be done, for example, by specifying $\mathbf{y}(t)$ at some point t_0.

If so, the problem we want to solve is

$$\begin{aligned}\dot{\mathbf{y}}(t) &= \mathbf{f}(t, \mathbf{y}),\\ \mathbf{y}(t_0) &= \mathbf{y}_0.\end{aligned}$$

This is an example of an *initial-value problem* (IVP).

The idea is to "march forward" with the information at time $t = t_0$ to produce approximations to the unknown function $\mathbf{y}(t)$ at some future final time $t = t_{\text{final}}$.

Note: There are other problems where not all of the missing information is specified at one point; rather it is *distributed* between 2 or more points; e.g., some components of $\mathbf{y}(t)$ could be specified at $t = t_0$ while others are specified at $t = t_{\text{final}}$.

Such problems are known as *boundary-value problems*.

Numerical methods for these problems differ markedly from those for initial-value problems, and we do not consider them here.

Modern numerical methods automatically determine the step sizes

$$h_n = t_{n+1} - t_n,$$

so that the estimated error in the numerical solution is controlled by a specified tolerance.

As usual, results from a single run of an algorithm on a problem should not be blindly trusted!

Rather, results from a few runs with decreasing tolerances should be compared to assess the accuracy of a given solution.

The Fundamental Theorem of Calculus gives us an important connection between differential equations and integrals:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + \int_t^{t+h} \mathbf{f}(s, \mathbf{y}(s))\, ds.$$

Note: The integral of a vector function simply means to take the integral of each component (similar to what we did for derivatives).

Unfortunately, we cannot use numerical quadrature directly to approximate the integral because we do not know the function $\mathbf{y}(s)$ in the integrand.

Nevertheless, the basic idea is to choose a sequence of values of h so that this formula allows us to generate our numerical solution.

One special case to keep in mind is the situation where $\mathbf{f}(t, \mathbf{y})$ is in fact a function of t alone.

Then indeed the numerical solution of such simple differential equations reduces to a sequence of quadratures:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_t^{t+h} \mathbf{f}(s)\, ds.$$

Of course, if we could compute the integral exactly, then we would obtain the exact solution to the ODE.

7.2 Systems of ODEs

Many mathematical models involve more than one unknown function; e.g., a model for a chemical reaction would probably have a differential equation for each chemical species.

Often mathematical models involve second- or higher-order derivatives; any model based on Newton's second law $F = ma$ corresponds to $\ddot{x}(t) = F/m$.

Problems such as these are not of the form $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, so it is not obvious how to use standard ODE software for their solution.

Fortunately, these models can be converted to first-order form using a standard change of variables.

In such cases, the unknowns are the function and its derivatives up to one order less than the given ODE.

This means that an mth-order ODE is converted to m first-order ODEs.

For example, consider the second-order differential equation describing a simple harmonic oscillator:

$$\ddot{x}(t) = -x(t).$$

In this case, we create the vector of unknown functions to be

$$y_1(t) = x(t), \qquad y_2(t) = \dot{x}(t).$$

Then the second-order ODE is converted into two first-order ODEs:

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ -y_1(t) \end{bmatrix}.$$

In order to use the built-in $\textsc{Matlab}$ routines to solve an IVP, we will have to input the right-hand side function $\mathbf{f}(t, \mathbf{y})$.

It should take as input a scalar t and a vector $\mathbf{y}(t)$ and return a vector $\mathbf{f}(t, \mathbf{y})$.

For the harmonic oscillator example, the MATLAB function defining $\mathbf{f}(t, \mathbf{y})$ might look like

```
function ydot = harmonic(t,y)
ydot = [y(2); -y(1)];
```

Notice that the variable t must be passed as the first argument, even though it is not involved in `ydot`.

As a slightly more complicated example, consider the *two-body problem*.

This describes the orbit of one body under the gravitational attraction of a much heavier body (like a planet orbiting a sun).

With the origin at the centre of the “sun”, the Cartesian coordinates $u(t)$ and $v(t)$ of the “planet” satisfy

$$\ddot{u}(t) = -\frac{u(t)}{[u^2(t) + v^2(t)]^{3/2}},$$
$$\ddot{v}(t) = -\frac{v(t)}{[u^2(t) + v^2(t)]^{3/2}}.$$

To convert to a first-order system of ODEs, we define

$$\mathbf{y}(t) = (u(t),\ \dot{u}(t),\ v(t),\ \dot{v}(t))^T.$$

Then

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \\ \dot{y}_3(t) \\ \dot{y}_4(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ -\frac{y_1(t)}{[y_1^2(t)+y_3^2(t)]^{3/2}} \\ y_4(t) \\ -\frac{y_3(t)}{[y_1^2(t)+y_3^2(t)]^{3/2}} \end{bmatrix}.$$

A MATLAB script to define this might look like

```
function ydot = twobody(t,y)
% define r for convenience and efficiency
r = (y(1)^2 + y(3)^2)^(3/2);
ydot = [y(2); -y(1)/r; y(4); -y(3)/r];
```

7.4 One-Step Methods

The simplest method for the numerical solution of IVPs is Euler's method[1].

Starting from a known value $\mathbf{y}_0$ at time $t = t_0$, it (typically) uses a fixed step size h and marches the approximate solution forward according to the formula

$$\begin{aligned}\mathbf{y}_{n+1} &= \mathbf{y}_n + h\,\mathbf{f}(t_n, \mathbf{y}_n),\\ t_{n+1} &= t_n + h, \qquad n = 0, 1, 2, \ldots.\end{aligned}$$

This method has a nice geometric interpretation.

We can imagine that $\mathbf{f}(t, \mathbf{y})$ specifies a velocity field for $\mathbf{y}$.

Then Euler's method can be thought of as sampling the velocity field at the point $(t_n, \mathbf{y}_n)$, assuming it is constant, and following it for a length of time h.

[1]Those in the biz also call it *forward Euler*.

This amounts to following the *tangent line* to the velocity field at $(t_n, \mathbf{y}_n)$ for some length of time h.

This procedure is repeated starting from $(t_{n+1}, \mathbf{y}_{n+1})$ to approximate $\mathbf{y}_{n+2}$, etc.

See forwardEulerDemo.m and forwardEulerDemo2.m

Given initial time t_0, initial value $\mathbf{y}_0$, final time t_{final}, right-hand side function $\mathbf{f}(t, \mathbf{y})$, and stepsize h, MATLAB code to implement Euler's method might look like:

```
t = t0;
y = y0;
while t <= tfinal
  y = y + h*feval(f,t,y)
  t = t + h
end
```

As you can imagine, the answer gets more accurate as you reduce h; but of course the more work has to be done to get to t_{final}.

Note: There is no inherent need to assume h is a constant.

In fact, if we could estimate how big a stepsize h_{n+1} starting from t_n we could get away with and still satisfy the given tolerance, then we could take that instead.

This generally yields a more efficient algorithm than one with a fixed step size.

As a quadrature rule for integrating $\mathbf{f}(t)$, Euler's method corresponds to left-hand rectangle rule; i.e., at each step, we take $\mathbf{f}(t) \approx \mathbf{f}(t_n)$.

A powerful way to analyze the accuracy of an IVP solver is by means of Taylor series.

Consider the Taylor series expansion for a scalar function $y(t+h)$ about the point $y(t)$:

$$\begin{aligned} y(t+h) &= y(t) + \dot{y}(t)\,h + \frac{1}{2}\ddot{y}(t)\,h^2 + \dots \\ &= y(t) + f(t, y(t))\,h + \frac{1}{2}\dot{f}(t, y(t))h^2 + \dots . \end{aligned}$$

One method to derive the forward Euler method is to truncate the Taylor series after terms of order h.

This leads (yet again) to

$$y_{n+1} = y_n + hf(t_n, y_n).$$

We define the order of accuracy of an IVP solver to be one order **less** than the order of the first term not matched in the Taylor series of the exact solution.

So, the forward Euler method is first-order accurate.

The idea is that although each step appears to be order h^2, in general there is an accumulation of errors from each step, and we assume there will be $\mathcal{O}(1/h)$ steps.

This just means that you probably have $t_{\mathsf{final}} = \mathcal{O}(1)$.

A first-order method is usually not very efficient: Tiny steps are needed to get even a few digits of accuracy.

It is easy to imagine that you could derive higher-order methods by keeping more terms in the Taylor series.

This is in fact true; the ensuing methods are (appropriately enough) called Taylor series methods.

The drawback of such approaches is that for anything beyond first order, you need to take derivatives of $\mathbf{f}(t, y(t))$ with respect to t.

Even for scalar equations, this involves the chain rule:

$$\begin{aligned}\dot{f}(t, y(t)) &= \frac{\partial f}{\partial t}(t, y(t)) + \frac{\partial f}{\partial y}(t, y(t))\dot{y}(t) \\ &= \frac{\partial f}{\partial t}(t, y(t)) + \frac{\partial f}{\partial y}(t, y(t))f(t, y(t)).\end{aligned}$$

This is already pretty messy, and the messiness is compounded exponentially as you take more derivatives.

Despite this, Taylor series methods have become more popular since the advent of symbolic algebra and automatic differentiation software packages.

However, we would like a way to obtain higher order only using evaluations of $\mathbf{f}(t, \mathbf{y})$.

Higher-order methods

There are 2 classical strategies to obtain methods with higher order:

1. Keep more past values of the solution; e.g., store

   $$\mathbf{y}_{n-i+1}, \quad \text{for } i = 1, 2, \ldots, k.$$

   This is called a k-step linear multistep method.

2. Build up approximations to $\mathbf{y}(t)$ at intermediate points in $[t_n, t_{n+1}]$. These are called Runge–Kutta methods after the two German applied mathematicians who first wrote about them in 1905.

- Multistep methods

A one-step numerical method has a short memory; it only uses information from one step of the integration.

As the name implies, a multistep method has a longer memory.

(In fact, an old-fashioned-sounding way to refer to them is methods with memory.)

The idea is that an interpolation is done on the past $\mathbf{y}$ and/or $\mathbf{f}$ values to derive a rule to approximate $\mathbf{y}_{n+1}$.

Multistep methods tend to be more efficient than one-step methods for problems with smooth solutions and high accuracy requirements.

For example, the orbits of planets and deep space probes are computed with multistep methods.

In production software, these methods vary both the order p and the step size h to achieve the most efficient integration.

However, natural questions such as how to start a multistep method (where do the past values come from at the beginning of the integration?) are beyond the scope of this course.

• Runge–Kutta methods

There are two natural possibilities for extending Euler's method by adding one function evaluation.

They correspond to the midpoint rule and the trapezoidal rule for quadrature.

For the midpoint rule, we step with Euler's method halfway across the interval, evaluate the function at this intermediate point, and then use that slope to take the actual step:

$$\begin{aligned}
\mathbf{K}_1 &= \mathbf{f}(t_n, \mathbf{y}_n) \\
\mathbf{K}_2 &= \mathbf{f}(t_n + h/2, \mathbf{y}_n + h\mathbf{K}_1/2) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{K}_2 \\
t_{n+1} &= t_n + h
\end{aligned}$$

For the trapezoidal rule, we (tentatively) step with Euler's method all the way across the interval, evaluate the function at this tentative point, and then average the two slopes to take the actual step:

$$\begin{aligned}
\mathbf{K}_1 &= \mathbf{f}(t_n, \mathbf{y}_n) \\
\mathbf{K}_2 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{K}_1) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + h(\mathbf{K}_1 + \mathbf{K}_2)/2 \\
t_{n+1} &= t_n + h
\end{aligned}$$

Perhaps not surprisingly, both of these methods are second-order accurate.

The most popular Runge–Kutta method has 4 stages (the number of $\mathbf{K}_i$) and is fourth-order accurate:

$$\begin{aligned}
\mathbf{K}_1 &= \mathbf{f}(t_n, \mathbf{y}_n) \\
\mathbf{K}_2 &= \mathbf{f}(t_n + h/2, \mathbf{y}_n + h\mathbf{K}_1/2) \\
\mathbf{K}_3 &= \mathbf{f}(t_n + h/2, \mathbf{y}_n + h\mathbf{K}_2/2) \\
\mathbf{K}_4 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{K}_3) \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + h(\mathbf{K}_1 + 2(\mathbf{K}_2 + \mathbf{K}_3) + \mathbf{K}_4)/6 \\
t_{n+1} &= t_n + h
\end{aligned}$$

If $\mathbf{f}(t, \mathbf{y})$ does not depend on $\mathbf{y}$, then the classical Runge–Kutta method has $\mathbf{K}_2 = \mathbf{K}_3$, and the method reduces to Simpson's quadrature rule.

Note: These methods are in fact explicit Runge–Kutta methods; i.e., the $\mathbf{K}_i$ are evaluated one after the other (like a recipe) and then a linear combination of them is added to $\mathbf{y}_n$ to obtain $\mathbf{y}_{n+1}$.

There are also implicit Runge–Kutta methods where all the $\mathbf{K}_i$ have to be solved at once as part of a big nonlinear system of equations.

Implicit Runge–Kutta methods are good for “stiff” (numerically tough) problems.

Despite being more work per step, high-order methods are much more efficient than low-order methods at producing very accurate answers to problems with smooth solutions.

Error Control

As usual, for the purposes of efficiency and reliability, we need to be able to change the stepsize h while satisfying a user-specified tolerance.

We now describe a method for controlling the error of the forward Euler method.

It is a similar strategy to that of adaptive quadrature: we take a step of size h, then 2 steps of size $h/2$, and compare the answers to estimate the error.

From this, we will also predict the optimal size of the next step.

To simplify the analysis, we assume that we are only dealing with 1 first-order ODE.

Suppose we wanted each step of forward Euler to satisfy a local error tolerance TOL.

We know the first term of the Taylor series that we neglected is $\mathcal{O}(h^2)$.

If we now take 2 steps of size $h/2$, we will have two approximations of the same quantity, and we can take a difference between them to estimate the error.

Analogously, consider the following strategy for error control of a scalar ODE with solution $y(t)$ by a method of order p:

Take 1 step of h to obtain the approximation y_h.

Take 2 steps of $h/2$ to obtain the approximation $y_{h/2}$.

Both y_h and $y_{h/2}$ are meant to approximate $y(t_n)$.

Then
$$\begin{aligned} y(t_n) - y_h &\approx c_h h^{p+1}, \\ y(t_n) - y_{h/2} &\approx 2c_{h/2}(h/2)^{p+1}. \end{aligned}$$

If we assume $c_h = c_{h/2} := c$,

$$\therefore \quad |y_h - y_{h/2}| \approx \left(1 - \tfrac{1}{2^p}\right) |c| h^{p+1},$$
$$\rightarrow \quad |c| h^{p+1} \approx \tfrac{2^p}{2^p - 1} |y_h - y_{h/2}|.$$

If this expression is less than the user's error tolerance TOL, the step with size h can be accepted as being sufficiently accurate.

In practice, we compare the error estimate to $frac{*}$TOL, where $frac$ is a fudge factor (usually taken to be ≈ 0.9) in order to err on the side of caution.

Dividing the above error estimate by 2^p allows us to have an error estimate for $y_{h/2}$.

We take $y_{h/2}$ to be the value with which we advance in practice.

Not only do we have error control, but we can also use this information to estimate the "optimal" stepsize for the next step.

We estimate what stepsize we could have taken to just satisfy the error tolerance.

This corresponds to the largest possible stepsize (and hence the most efficient integration).

Call this stepsize $\tilde{h}$.

Then
$$|c|\tilde{h}^{p+1} \approx frac * \mathsf{TOL},$$

and we can use

$$\left(\frac{\tilde{h}}{h}\right)^{p+1} \approx \frac{2^p - 1}{2^p} \frac{frac * \mathsf{TOL}}{|y_h - y_{h/2}|}$$

to calculate $\tilde{h}$.

7.7 Examples

The following classical example from astronomy provides us with strong motivation to solve ODEs with error control.

Consider the motion of a planet with a (normalized) mass of $\mu = 0.012277471$, a sun with mass $\hat{\mu} = 1 - \mu$, and a moon with negligible mass in a 2D plane.

The motion is governed by the equations

$$\begin{aligned}
\ddot{u}_1 &= u_1 + 2\dot{u}_2 - \hat{\mu}\frac{u_1 + \mu}{r_1} - \mu\frac{u_1 - \hat{\mu}}{r_2}, \\
\ddot{u}_2 &= u_2 - 2\dot{u}_1 - \hat{\mu}\frac{u_2}{r_1} - \mu\frac{u_2}{r_2}, \\
r_1 &= [(u_1 + \mu)^2 + u_2^2]^{3/2}, \\
r_2 &= [(u_1 - \hat{\mu})^2 + u_2^2]^{3/2}.
\end{aligned}$$

Starting with the initial conditions

$$u_1(0) = 0.994, \qquad u_2(0) = 0, \qquad \dot{u}_1(0) = 0,$$
$$\dot{u}_2(0) = -2.00158510637908252240537862224,$$

the orbit is periodic with period $T < 17.1$. Note that $r_1 = 0$ at $(u_1, u_2) = (-\mu, 0)$ and $r_2 = 0$ at $(u_1, u_2) = (\hat{\mu}, 0)$, so we need to be careful when the orbit comes close to these points!

We now solve this equation using Euler's method, the classical Runge–Kutta method, and MATLAB's `ode45`.

See `orbitSolvers.m`

7.8 Lorenz Attractor

One of the world's most famous ODE systems is the Lorenz chaotic attractor.

It was first described in 1963 by Edward Lorenz, a mathematician and meteorologist from MIT, who was interested in fluid flow models of the earth's atmosphere.

The ODEs themselves are

$$\begin{aligned}
\dot{y}_1 &= -\beta y_1 + y_2 y_3, \\
\dot{y}_2 &= -\sigma(y_2 - y_3), \\
\dot{y}_3 &= -y_1 y_2 + \rho y_2 - y_3.
\end{aligned}$$

$y_1(t)$ is related to the convection in the atmosphere;

$y_2(t)$, $y_3(t)$ are related to horizontal and vertical temperature variation.

The most popular values of the parameters, $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$ do not actually correspond to realistic values associated with the earth's atmosphere.

Theoretically of course the solution to the system is completely deterministic.

Nonetheless, the behaviour of the solution can appear to be very unpredictable.

(This is where the idea of chaos comes from.)

For some values of the parameters, the orbit of $\mathbf{y}(t)$ in 3D space is known as a strange attractor.

It is **bounded**, but **not periodic** and **not convergent**.

It never intersects itself.

It ranges chaotically back and forth around two different fixed points.

(For other values of the parameters, the solution might converge to a fixed point, diverge to infinity, or oscillate periodically.)

See `lorenzgui.m`

7.9 Stiffness

Stiffness is one of the most enigmatic yet pervasive concepts in the numerical solution of IVPs.

We look at one instance of a stiff problem: a model of flame propagation.

If you light a match, the ball of flame grows rapidly until it reaches a critical size.

Then it remains at that size because the amount of oxygen being consumed by the combustion in the interior of the ball balances the amount available through the surface.

The simple ODE model is

$$\dot{y} = y^2 - y^3, \quad y(0) = \delta, \quad 0 \leq t \leq 2/\delta.$$

The variable $y(t)$ represents the radius of the ball.

The y^2 and y^3 terms come from the surface area and the volume.

The critical parameter is the initial radius, δ, which is “small”.

We seek the solution over a length of time that is inversely proportional to δ.

We solve the problem for $\delta = 0.01$ using `ode23`.

Then we solve the problem for $\delta = 0.0001$ using `ode23`.

We see that when $y(t)$ reaches its equilibrium value, the integration slows to a crawl.

This is what is meant by stiffness: the numerical method takes a very small step-size even though the solution does not vary rapidly.

If instead we solve this problem with a stiff solver like `ode23s`, we see behaviour that is much more acceptable.

See `stiffnessDemo.m`