9 Initial value problems

In this chapter, we consider numerical methods for solving first order initial value problems (IVPs) of the form

$$\begin{aligned} y'(t) &= f(t,y),\ t_0 \le t \le T, && (9.1) \\ y(t_0) &= y_0, && (9.2) \end{aligned}$$

where t_0 is the start time, T is the endtime, and y_0 is the initial condition.

While students generally learn about certain types of IVPs in a sophomore differential equations course, they typically only learn about IVPs of very specific types, where exact solutions can be derived as closed form expressions. While theoretically this may be possible for many types of IVPs, it is unknown and maybe even not possible for most. Moreover, if one cannot find an exact solution in a book, deriving the exact solution one's self can be very difficult and time consuming, and may not be possible in a reasonable amount of time. The methods we discuss in this chapter approximate the solution at some finite number of t-points, and from this we can interpolate the values at every t in the interval $[t_0, T]$. That is, the 'solution' of a numerical ODE solver is a set of points

$$(t_0, y_0),\ (t_1, y_1),\ ...,\ (T, y_n).$$

For simplicity, we will consider equally spaced points, with spacing Δt.

It turns out that it is typically very easy to solve almost all IVPs using numerical methods. Solutions will be in numerical form (data points) instead of closed form expressions, but for most applications this is sufficient. One can always make an interpolant, i.e., using a cubic spline, if a solution between the discrete points is desired.

Note that even though we have written y and f as scalar functions, with all the methods we discuss below, they could also be considered vector functions. A first order system of IVPs has the form:

$$\begin{aligned} y_1'(t) &= f_1(t, y_1, \dots, y_n) \\ y_2'(t) &= f_2(t, y_1, \dots, y_n) \\ &\vdots \\ y_n'(t) &= f_n(t, y_1, \dots, y_n). \end{aligned}$$

We will show in the next section that solvers for this (seemingly simple) first order IVP above covers a very wide class of problems. This is because it covers vector systems, and higher order systems can usually be reduced to first order vector systems.

9.1 Reduction of higher order IVPs to first order

This section reviews that many higher order ODEs can be written as vector systems of first order ODEs. Recall that the order of an ODE is the highest number of derivatives in any of its terms. For example, the ODE

$$y'''(t) + y(t)y''(t) - t^2 = 0$$

is a third order ODE. Provided the ODE can be written in the form

$$y^{(n)}(t) = F(t, y, y', \dots, y^{(n-1)}),$$

then it can be written as a first order vector ODE by the following process:

– An nth order ODE will be turned into a first order ODE with n equations.

– Define functions $u_1,\ u_2, ..., u_n$ by $u_1(t) = y(t)$ and $u_i(t) = y^{(i-1)}(t)$ for $i = 2, 3, \dots, n$.

– The equations (identities) $u_i' = u_{i+1}$ for $i = 1, 2, ..., n-1$ form the first $n-1$ equations.

– For the last equation, use that $u_n' = y^{(n)}(t) = F(t, y, y', y'', \dots, y^{(n-1)}) = F(t, u_1, u_2, u_3, \dots, u_n)$.

Consider the following example.

Example 71. *Convert the following scalar IVP to a first order vector IVP:*

$$\begin{aligned} y'''(t) - ty''(t)y'(t) + y'(t) - ty(t) + \sin(t) &= 0, \\ y(t_0) &= y_0, \\ y'(t_0) &= z_0, \\ y''(t_0) &= w_0. \end{aligned}$$

The first step is to create the u functions, and since this is third order, we need 3 of them:

$$u_1 = y,\ u_2 = y',\ u_3 = y''.$$

We now have the three equations for the u functions:

$$\begin{aligned} u_1' &= u_2, \\ u_2' &= u_3, \\ u_3' &= tu_3u_2 - u_2 + tu_1 - \sin(t). \end{aligned}$$

Thus if we define

$$\mathbf{u}(t) = \begin{pmatrix} u_1(t) \\ u_2(t) \\ u_3(t) \end{pmatrix},$$

then we have that $\mathbf{u}'(t) = \mathbf{F}(\mathbf{u}, t)$*, with initial condition*

$$\mathbf{u}(t_0) = \begin{pmatrix} u_1(t_0) \\ u_2(t_0) \\ u_3(t_0) \end{pmatrix} = \begin{pmatrix} y(t_0) \\ y'(t_0) \\ y''(t_0) \end{pmatrix} = \begin{pmatrix} y_0 \\ z_0 \\ w_0 \end{pmatrix} = \mathbf{u}_0,$$

and

$$\mathbf{F}(t, \mathbf{u}) = \mathbf{F}(t, u_1, u_2, u_3) = \begin{pmatrix} u_2 \\ u_3 \\ tu_3u_2 - u_2 + tu_1 - \sin(t) \end{pmatrix}.$$

9.2 The forward Euler method

The most straight-forward numerical method to consider for IVPs is forward Euler. Integrating the initial value ODE (9.1) from t_i to t_{i+1} gives

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt. \tag{9.3}$$

Using the left rectangle rule to approximate the integral, we obtain the approximation

$$y(t_{i+1}) - y(t_i) \approx (t_{i+1} - t_i) f(t_i, y(t_i)).$$

Hence if we knew $y(t_i)$ (or some approximation of it), we could use this formula to approximate $y(t_{i+1})$. We now proceed by changing the approximation sign to an equals sign, and replace the true solution at t_i,$y(t_i)$ with its approximation y_i. This procedure can be formalized as follows.

Algorithm 72 (Forward Euler)**.** *Given a discretization in time* $\{t_0,\ t_1,\ t_2,\ ...,\ t_n = T\}$ *and initial condition* y_0*, find* $y_1,\ y_2,\ ...,\ y_n$ *by the following sequential process:*

$$y_{i+1} = y_i + \Delta t f(t_i, y_i)$$

Nice properties of forward Euler are that it is simple, easy to implement, and *explicit* (explicit in the sense that to calculate y_{i+1} you only need plug in known values t_i and y_i into the function f, and no equation needs to be solved). However, the method is not very accurate. This is perhaps not surprising, since we used the left rectangle quadrature approximation to create it. This quadrature rule is first order as a composite method, and so it is not surprising that forward Euler is also first order, in the sense of

$$\max_{1\le i\le n}\left|y(t_i)-y_i^{FE}\right|\le C\Delta t,$$

where $y(t_i)$ is the true solution at time t_i and y_i^{FE} is the forward Euler solution at time t_i. We prove this result now, after stating some preliminary lemmas about real numbers and sequences.

Lemma 73. *For a nonnegative sequence $\{a_i\}$ and nonnegative α and b, we have that*

$$a_{i+1}\le \alpha a_i+b \implies a_{i+1}\le \alpha^i a_0+\frac{\alpha^i-1}{\alpha-1}b.$$

Also, for any real x, $1+x\le e^x$, and for any real $x\ge -1$,

$$0\le (1+x)^m\le e^{mx}.$$

Proof. The proofs are left as exercises. The first is an elementary sequence property. For the second, use Taylor's theorem expanded about 0, and truncate at the quadratic term to show $1+x\le e^x$. The result for the power m then follows immediately. □

We now state a convergence theorem for forward Euler. As is typical throughout this book, we need to assume a sufficient number of derivatives exist for the true solution.

Theorem 74. *For a chosen $\Delta t>0$, consider the forward Euler approximation (with solution denoted by y_i^{FE} ($i=0,1,2,...,M=\frac{T}{\Delta t}$)) to the initial value problem $y'(t)=f(t,y)$ on $[0,T]$, with initial condition $y(0)=y_0$ (and $y_0^{FE}=y_0$). Assume the IVP solution has a bounded second derivative: $|y''(t)|\le K$, $\forall t\in[0,T]$, and that the function f is Lipschitz continuous in its second argument: there exists $F>0$ such that $|f(t,w)-f(t,z)|\le F|w-z|$ for any $t>0, w,z\in\mathbb{R}$. Then the error satisfies the bound*

$$\max_{1\le i\le M}|y(t_i)-y_i^{FE}|\le \frac{Ke^{TF}}{2F}\Delta t.$$

Proof. Begin by developing an error equation for $e_i : y(t_i) - y_i^{FE}$. The forward Euler approximation defines the y_i^{FE}'s by

$$\frac{y_{i+1}^{FE} - y_i^{FE}}{\Delta t} = f(t_i, y_i^{FE}), \tag{9.4}$$

and the true solution satisfies

$$y'(t_i) = f(t_i, y(t_i)).$$

Applying Taylor's theorem to the left hand side yields

$$\frac{y(t_{i+1}) - y(t_i)}{\Delta t} - \frac{\Delta t}{2} y''(t^*) = f(t_i, y(t_i)), \tag{9.5}$$

where t^* is fixed in $[t_i, t_{i+1}]$. Subtracting (9.4) from (9.5) gives the error equation

$$\frac{e_{i+1} - e_i}{\Delta t} = \frac{\Delta t}{2} y''(t^*) + \left(f(t_i, y(t_i)) - f(t_i, y_i^{FE})\right), \tag{9.6}$$

which reduces to

$$e_{i+1} = e_i + \frac{\Delta t^2}{2} y''(t^*) + \Delta t \left(f(t_i, y(t_i)) - f(t_i, y_i^{FE})\right). \tag{9.7}$$

We also note that $e_0 = 0$.

We now take absolute values of both sides, and then upper bound the terms on the right hand side with the triangle inequality and the assumptions on y'' and f. This gives the inequality

$$|e_{i+1}| \le |e_i| + \frac{K\Delta t^2}{2} + F\Delta t |e_i| = (1 + F\Delta t)|e_i| + \frac{K\Delta t^2}{2}. \tag{9.8}$$

Applying Lemma 73 to this inequality with $\alpha = (1 + F\Delta t)$ and $b = \frac{K\Delta t^2}{2}$, we obtain

$$\begin{aligned} |e_{i+1}| &\le (1 + F\Delta t)^i |e_0| + \frac{(1 + F\Delta t)^i - 1}{F\Delta t} \frac{K\Delta t^2}{2} \\ &\le \frac{K(1 + F\Delta t)^i}{2F} \Delta t, \end{aligned}$$

with the last step holding since $e_0 = 0$ and by reducing.

Applying the second part of Lemma 73, we have that

$$(1 + F\Delta t)^i \le e^{iF\Delta t} \le e^{M\Delta t F} = e^{TF},$$

since $T = M\Delta t$ and $0 \le i \le M$. Hence our error bound now becomes

$$|e_{i+1}| \le \frac{Ke^{TF}}{2F} \Delta t,$$

and since i was an arbitrary time step,

$$\max_{1\le i\le M} |e_i| \le \frac{Ke^{TF}}{2F}\Delta t.$$

This completes the proof. □

For numerical verification that the error is indeed first order, we consider the initial value problem

$$y'(t) = -1.2y + 7e^{-0.3t}, \quad y(0) = 3, \quad 0 \le t \le 2.5 = T,$$

which has the true solution $y(t) = \frac{70}{9}e^{-0.3t} - \frac{43}{9}e^{-1.2t}$. We compute solutions using Δt = T/10, T/20, T/40 and T/80, and calculate the max error at the node points. The errors are shown in the table below and are clearly first order.

Δt	$\max_i \lvert y(t_i) - y_i^{FE}\rvert$
$\frac{T}{10}$	**0.2610**
$\frac{T}{20}$	**0.1205**
$\frac{T}{40}$	**0.0580**
$\frac{T}{80}$	**0.0285**

9.3 Heun's method and RK4

As we have seen several times in this book, often with small modifications we can create much better methods. An obvious problem with forward Euler is that it is only first order accurate, and by now the reader is aware of how much more accurate a second (or higher) order method can be.

With this motivation, we consider why forward Euler is first order, and recall that the method was created by making an approximation using the left rectangle quadrature rule. We learned several more accurate quadrature approximations, and if we instead applied the trapezoidal rule, we obtain the IVP approximation

$$y(t_{i+1}) - y(t_i) \approx \frac{\Delta t}{2}\left(f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))\right).$$

This gives the approximation method known as the trapezoidal method:

$$y_{i+1} = y_i + \frac{\Delta t}{2}\left(f(t_i, y_i) + f(t_{i+1}, y_{i+1})\right).$$

As we expect, it can be proven that this method is $O(\Delta t^2)$ accurate. We omit the proof, but it follows similar to that of forward Euler above. However, using this method can be significantly more expensive than forward Euler because it is *implicit*; that is, to find y_{i+1}, it is not just a calculation to find y_i, instead an equation needs to be solved.

To obtain a more efficient method, we make one additional approximation. Recall that even though the forward Euler method is first order, from (9.8) we observe that as an approximation for a single time step, it is second order accurate (similar to the difference in quadrature approximation error on a single interval, compared to a composite rule). The German mathematician Heun noticed that we can have an *explicit, second order* method if we modify the trapezoidal method by replacing y_{i+1} with one step of a forward Euler approximation. Second order accuracy of the method is maintained, since we make a second order approximation inside of a second order method. The method is thus a 2 step method, as follows.

Algorithm 75 (Heun's method)**.** *Given a discretization in time* $t_0,\ t_1,\ t_2,\ ...,\ t_n = T$ *and initial condition* y_0*, find* $y_1,\ y_2,\ ...,\ y_n$ *by the following sequential process:*

$$\begin{aligned}\tilde{y}_{i+1} &= y_i + \Delta t f(t_i, y_i)\\ y_{i+1} &= y_i + \frac{\Delta t}{2}\left(f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})\right)\end{aligned}$$

Heun's method is twice as expensive as forward Euler, since here there are two function evaluations at each time step. However this is still far more efficient than a method that needs to do a (likely nonlinear) solve at each time step.

One can continue the idea of using better quadrature formulas and better approximations to derive even higher order IVP approximation methods. The most commonly used one is fourth order Runge Kutta, often called "RK4". It is derived by applying the Simpson quadrature formula to (9.3), then making approximations in a clever way. It is defined in four explicit steps:

Algorithm 76 (RK4)**.** *Given a discretization in time* $\{t_0,\ t_1,\ t_2,\ ...,\ t_n = T\}$ *and initial condition* y_0*, let* Δt *be the constant time step size, and find*

y_1, y_2, ..., y_n *by the following sequential process:*

$$\begin{aligned} K_1 &= f(t_i, y_i) \\ K_2 &= f\left(t_i + \frac{\Delta t}{2}, y_i + \frac{\Delta t}{2}K_1\right), \\ K_3 &= f\left(t_i + \frac{\Delta t}{2}, y_i + \frac{\Delta t}{2}K_2\right), \\ K_4 &= f\left(t_i + \Delta t, y_i + \Delta t K_3\right), \\ y_{i+1} &= y_i + \Delta t\left(\frac{K_1}{6} + \frac{K_2}{3} + \frac{K_3}{3} + \frac{K_4}{6}\right). \end{aligned}$$

RK4 is explicit and therefore very fast, and it is $O(\Delta t^4)$ accurate, which is significantly more accurate than the other methods above.

To compare these higher order methods to forward Euler, we repeat the test done above with forward Euler. Again, error is the max absolute difference at the nodes between the true solution and approximation method.

Δt	**forward Euler error**	**Heun's method error**	**RK4 error**
$\frac{T}{10}$	**2.6104e-01**	**2.6893e-02**	**1.2804e-04**
$\frac{T}{20}$	**1.2046e-01**	**5.9284e-03**	**7.0050e-06**
$\frac{T}{40}$	**5.8042e-02**	**1.3935e-03**	**4.0967e-07**
$\frac{T}{80}$	**2.8516e-02**	**3.3792e-04**	**2.4773e-08**

From the table of errors above, observe first that forward Euler is first order (errors are cut in half as Δt is halved), Heun's method is second order (errors are cut in fourth as Δt is halved), and RK4 is fourth order (errors are cut in sixteenth as Δt is halved). But most importantly, observe how RK4 is dramatically more accurate than the other two methods.

9.4 Stiff problems and numerical stability

Although the methods above work well on most types of problems, there are certain IVPs they do not work well on, in particular, those with steep slopes in their solutions. Such ODEs are called 'stiff' and can sometimes lead to numerical solvers exhibiting exponential growth to infinity, even though the true solution does not behave this way (in other words, your solution is so terrible it cannot even be considered an approximation). The reason for this is that if the time step Δt is too large, the solver will not correctly handle the steep slope, which leads to numerical instability.

Consider the following example, to illustrate this behavior:

Example 77. *A chemical decays proportional to its concentration to the 1.5 power, and simultaneously it is being produced. The ODE for its concentration is given by*

$$y' = -0.8 * y^{1.5} + 20000(1 - e^{-3t}).$$

Given $y(0) = 2000$*, approximate* y *on* $[0, 0.5]$ *using forward Euler.*

For $\Delta t = 0.1$ *and* 0.05*, we get exponential blowup (the* $t = 0.5$ *solutions are larger than* 10^8*). However, for* Δt *sufficiently small, we can obtain the correct solution, see the figure below.*

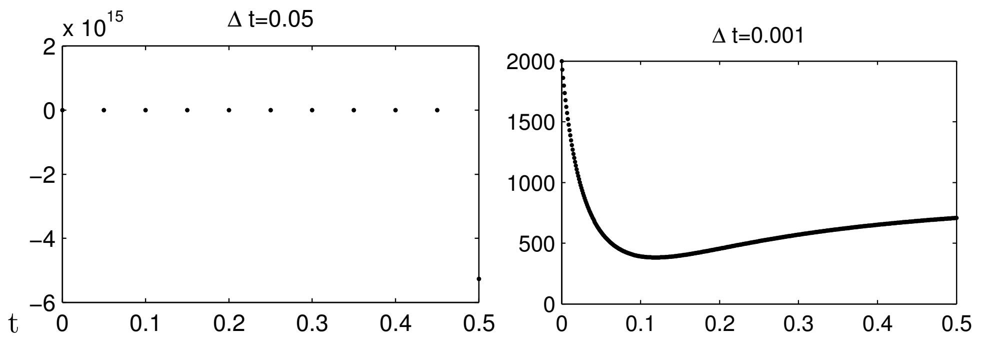


Fig. 9.1: Plots of forward Euler solution with $\Delta t = 0.05$ (left) and $\Delta t = 0.001$ (right).

The plots we get are shown in Figure 9.1. It is clear to see (look at the scale of the y-axis) that we experience unstable behavior (blow up to infinity) for $\Delta t = 0.1$ *and* $\Delta t = 0.05$ *solutions. Even though forward Euler converges as* $\Delta t \to 0$*, until* Δt *is small enough for it to be stable, the answer is no good. Once* Δt *is small enough, then we see convergence to the solution.*

That forward Euler (and all explicit methods, for that matter) requires Δt to be small enough to avoid catastrophic blow-up can be observed directly from some mathematical analysis. Consider the simple IVP

$$y' = \lambda y, \quad y(0) = y_0,$$

which has solution $y(t) = y_0 e^{\lambda t}$. If $\lambda < 0$, then this IVP solution will exhibit exponential decay. Hence, it would be a reasonable request to insist that an IVP solver does not exhibit exponential growth when the true solution decays exponentially. We note that even though this is a simple IVP, other IVPs will locally behave like it, since linear approximations are very good local approximations of functions.

Applying forward Euler to this simple IVP, we obtain

$$y_{n+1} = y_n + \Delta t f(t_n, y_n) = y_n + \Delta t \lambda y_n = (1 + \lambda \Delta t) y_n = ... = (1 + \lambda \Delta t)^{n+1} y_0.$$

Hence forward Euler yields a solution that decays exponentially only if $|1+\lambda\Delta t| \leq 1$, i.e. if $|\lambda|\Delta t < 2$, but otherwise grows exponentially. For this reason, we say **forward Euler is conditionally stable**. A similar but more complicated and technical analysis reveals that RK4 exhibits exponential decay only if

$$\left| 1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2 + \frac{1}{6}(\lambda\Delta t)^3 + \frac{1}{24}(\lambda\Delta t)^4 \right| \leq 1.$$

This is a similar conditional stability on Δt as for forward Euler, and hence we say **RK4 is conditionally stable.**

9.4.1 Implicit methods and unconditional stability

For most problems, the stability condition on Δt is reasonable, and RK4 can be used successfully. However, many problems exist where Δt needs to be so small for stability that one may never get an answer because so many time steps need to be taken. In cases such as this, we turn to *implicit* methods, which are more expensive than explicit methods at each step, but typically have little or no restriction on Δt for stability.

Consider a method called backward Euler, which is almost identical to forward Euler except that the right rectangle quadrature approximation is used instead of the left rectangle rule. This method takes the following form.

Algorithm 78 (backward Euler)**.** *Given a discretization in time* $t_0,\ t_1,\ t_2,\ ...,\ t_n = T$ *and initial condition* y_0*, find* $y_1,\ y_2,\ ...,\ y_n$ *by the following sequential process:*

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

The obvious disadvantage of backward Euler is that it is an *implicit* method, as at each time step we must perform a (probably nonlinear) solve to find y_{i+1} (in forward Euler there is no solve – just an explicit calculation). Hence at a glance, one might immediately dismiss this method as not competitive. However, this method has a special property shared by most implicit methods, which is that it has much better stability than explicit methods.

Consider backward Euler applied to the same simplified IVP above, again with $\lambda < 0$ (so the IVP solution decays exponentially). This yields

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

and so

$$y_{n+1} = (1 - \lambda \Delta t)^{-1} y_n = ... = (1 - \lambda \Delta t)^{-(n+1)} y_0.$$

Since $\lambda < 0$, we have that $\left| \frac{1}{1-\lambda \Delta t} \right| < 1$ for any choice of Δt. This means that **backward Euler is unconditionally stable**, as it will never exhibit exponential growth if the underlying IVP does not.

If we repeat the above example with backward Euler, we do not observe blowup for larger Δt as we did in forward Euler, see Figure 9.2. Of course, using too large of a time step will not give good results, but in no case will one get exponential growth for an approximation of a stable ODE.

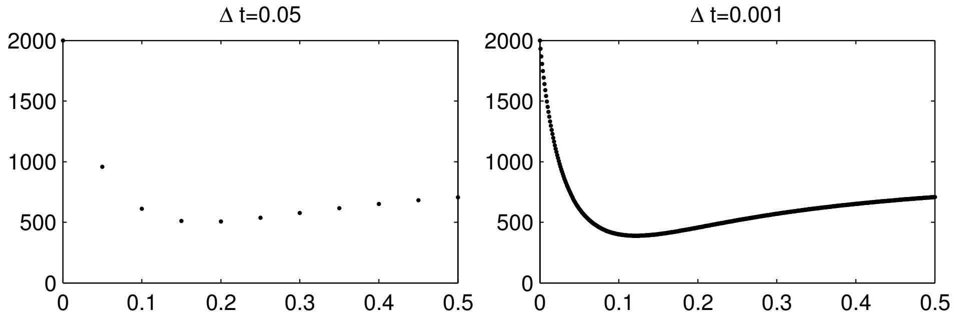


Fig. 9.2: Plots of backward Euler solutions with $\Delta t = 0.05$ (left) and $\Delta t = 0.001$ (right).

Since backward Euler is derived from a first order quadrature approximation, its error is only first order (showing this is an exercise). Similar to the case of explicit methods, we can derive more accurate implicit methods by better approximations. For example, the trapezoidal method above (before Heun's approximation is applied) is an implicit, unconditionally stable, second order method (showing this is an exercise). Higher order solvers can also be created, and although any method that has higher than second order will not be unconditionally stable, the stability conditions for implicit methods are generally much better than explicit methods.

9.5 Summary, general strategy, and MATLAB ODE solvers

The strategy for solving IVPs is actually quite simple. First try a high order explicit solver like RK4; if it works, you are done. If it fails, this is generally due to stability issues, and so you should resort to an implicit method. Of course, there are many more methods for solving ODEs than the four above. But now that we understand the basics, in most cases we can simply use the existing ODE solvers in MATLAB.

The MATLAB solver ‘ode45’ is based on RK4, but has numerous improvements built into it. For example, it shrinks the time step as necessary to get stability and a desired accuracy. Hence ‘ode45‘ is still an explicit (fast) and highly accurate solver provided the time step restriction for stability is not too restrictive. When this method fails, generally it does not finish (MATLAB just hangs and never gives an answer) - this is because Δt is so small that the method has not finished since so many time steps must be taken.

For ‘stiff’ problems where stability can still be an issue with ‘ode45‘, the solver ‘ode15s‘ should be tried next. It is an implicit solver that switches between different implicit methods so that it can achieve optimal accuracy while remaining stable. Note that for any implicit method, a solve has to be done, and so MATLAB lets you provide the derivative/Jacobian of the right hand side function f. If you don’t provide it, MATLAB will use finite differences, which in some cases can be inaccurate enough to cause the method to fail.

Let’s do an example:

Example 79. *Solve the initial value problem on* $1 \leq t \leq 2$*:*

$$\begin{aligned} t^3y''' - t^2y'' + 3ty' - 4y &= 5t^3 \ln t + 9t^2 \\ y(1) &= 0 \\ y'(1) &= 1 \\ y''(1) &= 3 \end{aligned}$$

First, we convert the ODE to first order. This gives the system

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}'(t) = \mathbf{u}'(t) = \mathbf{g}(t, \mathbf{u}) = \begin{pmatrix} u_2 \\ u_3 \\ t^{-1}u_3 - 3t^{-2}u_2 + 4t^{-3}u_1 + 5\ln t + 9t^{-1} \end{pmatrix}$$

with initial condition

$$\mathbf{u}(1) = \begin{pmatrix} 0 \\ 1 \\ 3 \end{pmatrix}.$$

Note that dividing through by the t^3 was permissible since it is never 0 on the domain.

Next, we create a right hand side function f to pass into MATLAB.

```
function f = odefun3(t,u)

f(1,1) = u(2);
f(2,1) = u(3);
f(3,1) = u(3)/t − 3*u(2)/(t^2) + 4*u(1)/(t^3) + 5*log(t) + 9/t;
```

The ode solver can then be called with

```
>> [t,y] = ode45(@odefun3,[1 2],[0 1 3]);
```

Notice that we only gave MATLAB the start and end time, we did not discretize t. MATLAB uses its defaults to pick each individual timestep, and they are not (usually) chosen uniformly. In this case, there were 57 t points chosen

```
>> size(t)

ans =

    57     1
```

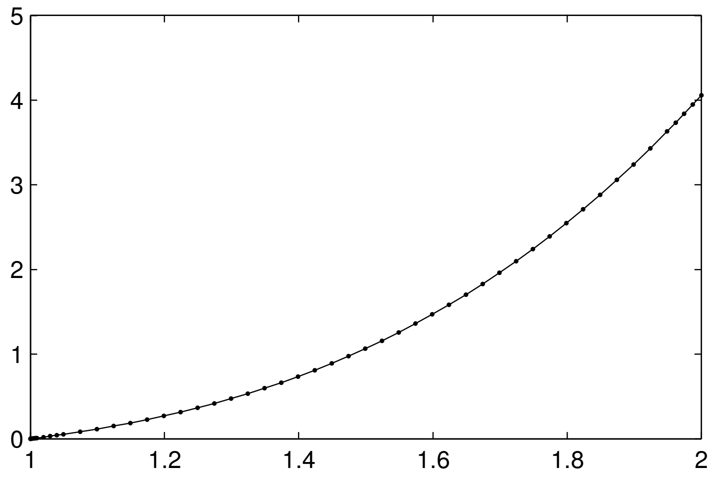
However, we are allowed to increase the desired accuracy with the odeset options. The default absolute error tolerance is 1e-6, and if we make it smaller, then more t points are used.

```
>> options = odeset('AbsTol',1e−10);
>> [t,y] = ode45(@odefun3,[1 2],[0 1 3],options);
>> size(t)

ans =

    81     1
```

The solution is held in y, but note that y holds all 3 components: $[y, y', y'']$. If we want to plot just y, we say

```
>> plot(t,y(:,1),'k.−')
```

to produce the plot



9.6 Fitting ODE parameters to data

It is common in many applications that problem/material parameters are determined from using data together with differential equations. That is, we may know the differential equation that governs a system, except for some parameters which are difficult to obtain or measure. However, we may also know data points that can help us determine what the parameters are.

Consider first an unforced mass-spring system, which is governed by the second order initial value problem

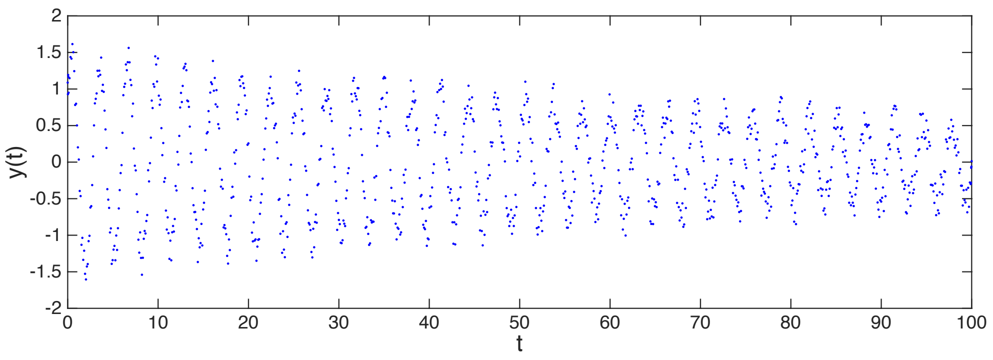
$$my''(t) + dy'(t) + ky(t) = 0,$$

where m is the mass, d is a damping coefficient, and k is the spring constant. We reduce the number of parameters to two by dividing through by m, which gives the system

$$y''(t) + \alpha y'(t) + \beta y(t) = 0.$$

Since m is easy to measure, we can easily recover d and k once we determine α and β.

Consider an example where we know $y(0) = 1$, $y'(0) = 2$, $m = 100$, and we have the following data points (t_i, y_i), $i = 1, 2, ..., n$ (which were generated by adding random noise $unif(-.25, .25)$ to a system with $c = 2$ and $k = 400$).



The plot above was generated with the commands

```
a=1;
b=2;
m=100;
c = 2;
k = 400;

alpha=c/m;
beta = k/m;

% setup and solve the ode
oderhs = @(t,y) [ y(2); -alpha*y(2)-beta*y(1)];
ode0 = [a;b];
[t,u] = ode45(oderhs,[0 100],ode0);

% randomly perturb the data
u2 = u(:,1) + .5*(rand( size(u,1),1)-0.5);

tdata=t;
ydata=u2;

plot(tdata,ydata,'b.')
hold on
xlabel('t','FontSize',20)
ylabel('y(t)','FontSize',20)
set(gca,'FontSize',16)
```

Now assuming this data is from measurements (i.e. assuming we don't know where it came from), we wish to determine α and β to minimize the sum of squares error, which we define to be

$$e(\alpha,\beta) = \sum_{i=1}^{n} \left(y(\alpha,\beta,t_i) - y_i\right)^2 ,$$

where $y(\alpha, \beta, t_i)$ is the solution of the ODE with parameters α and β, evaluated at $t = t_i$ (using a spline approximation of $y_{\alpha\beta}$ if t_i is not a point used by the ODE solver).

To solve the IVP, we note that since it is second order, we must convert it to first order via $u_1 = y$, $u_2 = y'$, so that

$$\mathbf{u}'(t) = \begin{pmatrix} u_2 \\ \\ -\alpha u_2 - \beta u_1 \end{pmatrix}, \quad \mathbf{u}(0) = \begin{pmatrix} 1 \\ \\ 2 \end{pmatrix}$$

The function below inputs the parameters α and β, the data points, and the initial condition $(a, b)^T$, and returns the sum of squares error.

```
function err = mass_spring_error(alpha,beta,tdata,ydata,a,b)
% function err = mass_spring_error(alpha,beta,tdata,ydata,a,b)

% input: parameters alpha and beta
% input: data points - tdata, ydata
% input: initial condition y(0)=a, y'(0)=b
% output: err (the sum of squares error)

% create ODE pieces
oderhs = @(t,y) [ y(2); -alpha*y(2)-beta*y(1)];
ode0 = [a;b];

% solve ivp
[t,u] = ode45(oderhs,[0 100],ode0);

% make a spline of the ivp solution
cs = spline(t,u(:,1));

% compare data points to this spline
err = sum( ( ydata - ppval(cs,tdata) ).^2);
```

Now that we have defined an error function in terms of α and β, we can use MATLAB's built-in optimization routines to find the optimal α and β (i.e. the α and β that give the smallest error). Optimization is an important topic in Mathematics, but it is not in the scope of this book or course, and so we will use the routines without discussing how they work. 'fminsearch' is a MATLAB routine that inputs an error function (which can be vector valued), an initial guess at the optimal value(s), and returns an optimal or quasi-optimal solution. We use it as follows, with m being the vector of unknowns and $(\alpha, \beta) \approx (.1, 2)$ as the initial guess (an initial guess can be found by 'playing with' α and β in the ODE solver, trying to get a curve somewhat close to the data).

```
err_total = @(m) ( mass_spring_error(m(1),m(2),tdata,ydata,a,b) );
[params,val,flag,output] = fminsearch(err_total,[.1;2]);
```

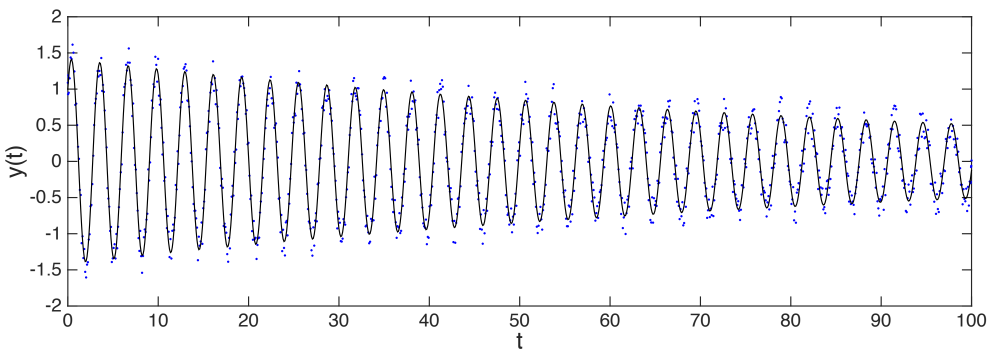
The routine is vector based, and the optimal α and β are stored in the vector 'params'. Thus, we can recover them, plot the solution with the data, and recover the k and c parameter values (since we know m=100).

```
alpha_fitted=params(1);
beta_fitted=params(2);

oderhs_fitted = @(t,y) [ y(2); -alpha_fitted*y(2)-beta_fitted*y(1)];
ode0 = [a;b];

[t,u] = ode45(oderhs_fitted,[0 100],ode0);
plot(t,u(:,1),'k-')

% we know m=100, we can now back out c and k
c_fitted = alpha_fitted*m
k_fitted = beta_fitted*m
```

This yields the the parameter estimates

```
c_fitted =

   2.028000895720333

k_fitted =

     4.000723693572518e+02
```

which we know are quite accurate, since we generated the (very) noisy data with $c = 2$ and $k = 400$. The code above also generates the following plot of the optimal ODE solution (plotted together with the data points).



9.7 Exercises

1. Consider the IVP on $[1, 2]$:
$$y'(t) = ty^2,$$
with initial condition $y(1) = 1$. Using $\Delta t = 0.5$, approximate *by hand* the solution $y(t)$ at $t = 2$ using forward Euler, Heun's method, RK4, and backward Euler. You can use MATLAB or a calculator to do individual calculations, but you are not to just run a MATLAB solver.
2. Consider the following initial value problem on $0 \le t \le 2$:
$$y'(t) = e^t(\sin(t) + \cos(t)) + 2t, \quad y(0) = 1.$$
Calculate solutions to this problem using forward Euler, backward Euler, the trapezoidal method, and RK4, each with varying discretizations $\Delta t = \frac{1}{10}, \frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}$. Create a table of errors and convergence rates, noting that the true solution is
$$y(t) = e^t \sin(t) + t^2 + 1.$$
Do your estimated convergence rates match the theory for each method?
3. Solve the following initial value problem using 'ode45' and 'ode15s':
$$y'''(t) - 3y''(t) + ty(t) - \sin^2(t) = 7, \; 0 \le t \le 1, \; y(0) = 0, \; y'(0) = 1, \; y''(0) = 0.$$
Plot the solution for varying tolerances. Why do you believe your solution is correct?
4. Consider the initial value problem
$$y'(t) = -1.2y + 7e^{-0.3t}, \quad y(0) = 3, \quad 0 \le t \le 2.5 = T.$$
Using Δt =T/1000, solve the problem using forward Euler, backward Euler, and RK4. Time each computation using tic/toc (run each one a couple times, just to make sure your computer does not have other processes running that might slow it down). Explain what you observe.
5. Consider the initial value problem on $0 \le t \le 3$:
$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}' = \begin{pmatrix} -4u_1 + 2u_2 + \cos(t) + 4\sin(t) \\ 3u_1 + u_2 - 3\sin(t) \end{pmatrix}$$

The initial condition is $u_1(0) = 1,\ u_2(0) = 1$. Solve this ODE using MATLAB's 'ode15s' and 'ode45s'. Plot your solution. Why do you believe your solution is correct?

6. Water flows out of an inverted conical talk with circular orifice at the rate of

$$y'(t) = -.7\pi r^2\sqrt{-2g}\frac{\sqrt{y}}{A(y)}$$

where r is the orifice radius, y is the height of the water above the orifice, and A(y) is the area of the cross section of the tank at the water level. Suppose $r = .1$ ft, $g = -32.17$ ft/s, and the tank has an initial water level of 8 ft and initial volume of $512(\pi/3)$ ft^3. Use 'ode45' to approximate a solution, and use it to determine when the tank will be empty.

7. Consider the initial value problem

$$y'(t) = -1.2y + 7e^{-0.3t},\quad y(0) = 3,\quad 0 \le t \le 2.5 = T,$$

which has true solution

$$y(t) = \frac{70}{9}e^{-0.3t} - \frac{43}{9}e^{-1.2t}.$$

For each of the 4 methods: forward Euler, backward Euler, Heun's, and RK4 compute solutions using Δt = T/10, T/20, T/40 and T/80, and calculate the error

$$err = \max_{i=1,2,\ldots,n}\left| y_{true}(t_i) - y_i \right|$$

for each test. From these errors, calculate convergence rates for the 4 methods.

8. Using the ODE from Example 77, compare the stability of the methods: forward Euler, backward Euler, trapezoidal, Heun's, and RK4. What is the Δt needed for each method to be stable (i.e. not grow exponentially)?

9. Prove that

$$a_{i+1} \le \alpha a_i + b \implies a_{i+1} \le \alpha^i a_0 + \frac{\alpha^i - 1}{\alpha - 1}b.$$

10. Prove Lemma 73.

11. Prove an error estimate for backward Euler, similar to Theorem 74's result forward Euler.

12. Consider the implicit, trapezoidal method for solving IVPs:

$$y_{n+1} = y_n + \Delta t f\left(\frac{t_n + t_{n+1}}{2}, \frac{y_n + y_{n+1}}{2}\right).$$

Apply the method to the IVP $y' = \lambda y,\ y(0) = y_0$ with $\lambda < 0$ to show the method is unconditionally stable, and prove that the error in the method is $O(\Delta t^2)$.

13. Consider the population data

```
data =[
1790    3.929
1800    5.308
1810    7.240
1820    9.638
1830    12.866
1840    17.069
1850    23.192
1860    31.443
1870    38.558
1880    50.156
1890    62.948
1900    75.996
1910    91.972
1920    105.711
1930    122.775
1940    131.669
```

Often for population estimation, logistic ODEs are considered, which take the form

$$y'(t) = \alpha y(\beta - y) \quad \text{for } t > 0, \quad y(0) = a.$$

This problem is an exercise, so ignore the fact that you could solve this ODE in closed form.

Your task is to best fit (by minimizing least square error) the α and β to the data. Note that numerically, it may be best to shift time by subtracting 1790 off of each t point. A very rough guess of $\alpha = 0.0001$ and $\beta = 200$ will be sufficient for the optimization routine to converge.

Use your newly found 'optimal' ODE to predict the population until 1990. The actual data is as follows.

```
1950    150.697
1960    179.323
1970    203.185
1980    226.546
1990    248.710
```

Comment on the ability of this "optimal" solution to predict the future.