Chapter 8

Dynamics and Differential Equations

Many dynamical systems are modeled by first order systems of differential equations. An n component vector $x(t) = (x_1(t), \ldots, x_n(t))$, models the state of the system at time t. The dynamics are modelled by

$$\frac{dx}{dt} = \dot{x}(t) = f(x(t), t) \ , \tag{8.1}$$

where $f(x,t)$ is an n component function, $f(x,t) = f_1(x,t), \ldots, f_n(x,t))$. The system is *autonomous* if $f = f(x)$, i.e., if f is independent of t. A *trajectory* is a function, $x(t)$, that satisfies (8.1). The *initial value problem* is to find a trajectory that satisfies the *initial condition*[1]

$$x(0) = x_0 \ , \tag{8.2}$$

with x_0 being the *initial data*. In practice we often want to specify initial data at some time other than $t_0 = 0$. We set $t_0 = 0$ for convenience of discussion. If $f(x,t)$ is a differentiable function of x and t, then the initial value problem has a solution trajectory defined at least for t close enough to t_0. The solution is unique as long as it exists.[2].

Some problems can be reformulated into the form (8.1), (8.2). For example, suppose $F(r)$ is the force on an object of mass m if the position of the object is $r \in R^3$. Newton's equation of motion: $F = ma$ is

$$m\frac{d^2r}{dt^2} = F(r) \ . \tag{8.3}$$

This is a system of three second order differential equations. The velocity at time t is $v(t) = \dot{r}(t)$. The trajectory, $r(t)$, is determined by the initial position, $r_0 = r(0)$, and the initial velocity, $v_0 = v(0)$.

We can reformulate this as a system of six first order differential equations for the position and velocity, $x(t) = (r(t), v(t))$. In components, this is

$$\begin{pmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ x_5(t) \\ x_6(t) \end{pmatrix} = \begin{pmatrix} r_1(t) \\ r_2(t) \\ r_3(t) \\ v_1(t) \\ v_2(t) \\ v_3(t) \end{pmatrix} \ .$$

[1] There is a conflict of notation that we hope causes little confusion. Sometimes, as here, x_k refers to component k of the vector x. More often x_k refers to an approximation of the vector x at time t_k.

[2] This is the existence and uniqueness theorem for ordinary differential equations. See any good book on ordinary differential equations for details.

The dynamics are given by $\dot{r} = v$ and $\dot{v} = \frac{1}{m}F(r)$. This puts the equations (8.3) into the form (8.1) where

$$f = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} = \begin{pmatrix} x_4 \\ x_5 \\ x_6 \\ \frac{1}{m}F_1(x_1, x_2, x_3) \\ \frac{1}{m}F_2(x_1, x_2, x_3) \\ \frac{1}{m}F_3(x_1, x_2, x_3) \end{pmatrix} \quad .$$

There are variants of this scheme, such as taking $x_1 = r_1$, $x_2 = \dot{r}_1$, $x_3 = r_2$, etc., or using the momentum, $p = m\dot{r}$ rather than the velocity, $v = \dot{r}$. The initial values for the six compenents $x_0 = x(t_0)$ are given by the initial position and velocity components.

8.1 Time stepping and the forward Euler method

For simplicity this section supposes f does not depend on t, so that (8.1) is just $\dot{x} = f(x)$. *Time stepping*, or *marching*, means computing approximate values of $x(t)$ by advancing time in a large number of small steps. For example, if we know $x(t)$, then we can estimate $x(t + \Delta t)$ using

$$x(t + \Delta t) \approx x(t) + \Delta t \dot{x}(t) = x(t) + \Delta t f(x(t)) \; . \tag{8.4}$$

If we have an approximate value of $x(t)$, then we can use (8.4) to get an approximate value of $x(t + \Delta t)$.

This can be organized into a method for approximating the whole trajectory $x(t)$ for $0 \leq t \leq T$. Choose a *time step*, Δt, and define discrete times $t_k = k\Delta t$ for $k = 0, 1, 2, \ldots$. We compute a sequence of approximate values $x_k \approx x(t_k)$. The approximation (8.4) gives

$$x_{k+1} \approx x(t_{k+1}) = x(t_k + \Delta t) \approx x(t_k) + \Delta t f(x_k) \approx x_k + \Delta t f(x_k) \; .$$

The *forward Euler* method takes this approximation as the definition of x_{k+1}:

$$x_{k+1} = x_k + \Delta t f(x_k) \quad . \tag{8.5}$$

Starting with initial condition x_0 (8.5) allows us to calculate x_1, then x_2, and so on as far as we like.

Solving differential equations sometimes is called integration. This is because of the fact that if $f(x, t)$ is independent of x, then $\dot{x}(t) = f(t)$ and the solution of the initial value problem (8.1) (8.2) is given by

$$x(t) = x(0) + \int_0^t f(s) ds \; .$$

If we solve this using the rectangle rule with time step Δt, we get

$$x(t_k) \approx x_k = x(0) + \Delta t \sum_{j=0}^{k-1} f(t_j) \; .$$

We see from this that $x_{k+1} = x_k + \Delta t f(t_k)$, which is the forward Euler method in this case. We know that the rectangle rule for integration is first order accurate. This is a hint that the forward Euler method is first order accurate more generally.

We can estimate the accuracy of the forward Euler method using an informal *error propagation equation*. The error, as well as the solution, evolves (or propagates) from one time step to the next. We write the value of the exact solution at time t_k as $\widetilde{x}_k = x(t_k)$. The error at time t_k is $e_k = x_k - \widetilde{x}_k$. The *residual* is the amount by which $\widetilde{x}_k$ fails to satisfy the forward Euler equations[3] (8.5):

$$\widetilde{x}_{k+1} = \widetilde{x}_k + \Delta t f(\widetilde{x}_k) + \Delta t r_k \ , \tag{8.6}$$

which can be rewritten as

$$r_k = \frac{x(t_k + \Delta t) - x(t_k)}{\Delta t} - f(x(t_k)) \ . \tag{8.7}$$

In Section 3.2 we showed that

$$\frac{x(t_k + \Delta t) - x(t_k)}{\Delta t} = \dot{x}(t_k) + \frac{\Delta t}{2}\ddot{x}(t_k) + O(\Delta t^2) \ .$$

Together with $\dot{x} = f(x)$, this shows that

$$r_k = \frac{\Delta t}{2}\ddot{x}(t_k) + O(\Delta t^2) \ , \tag{8.8}$$

which shows that $r_k = O(\Delta t)$.

The error propagation equation, (8.10) below, estimates e in terms of the residual. We can estimate $e_k = x_k - \widetilde{x}_k = x_k - x(t_k)$ by comparing (8.5) to (8.6)

$$e_{k+1} = e_k + \Delta t \left(f(x(k)) - f(\widetilde{x}_k) \right) - \Delta t r_k \ .$$

This resembles the forward Euler method applied to approximating some function $e(t)$. Being optimistic, we suppose that x_k and $x(t_k)$ are close enough to use the approximation (f' is the Jacobian matrix of f as in Section **??**)

$$f(x_k) = f(\widetilde{x}_k + e_k) \approx f(\widetilde{x}_k) + f'(\widetilde{x}_k) e_k \ ,$$

and then

$$e_{k+1} \approx e_k + \Delta t \left(f'(\widetilde{x}_k) e_k - r_k \right) \ . \tag{8.9}$$

If this were an equality, it would imply that the e_k satisfy the forward Euler approximation to the differential equation

$$\dot{e} = f'\left(x\left(t\right)\right) e - r(t) \ , \tag{8.10}$$

where $x(t)$ satisfies (8.1), e has initial condition $e(0) = 0$, and $r(t)$ is given by (8.8):

$$r(t) = \frac{\Delta t}{2}\ddot{x}(t) \ . \tag{8.11}$$

[3] We take out one factor of Δt so that the order of magnitude of r is the order of magnitude of the error e.

The *error propagation equation* (8.10) is linear, so $e(t)$ should be proportional to[4] r, which is proportional to Δt. If the approximate $e(t)$ satisfies $\|e(t)\| = C(t)\Delta t$, then the exact $e(t)$ should satisfy $\|e(t)\| = O(\Delta t)$, which means there is a $C(t)$ with

$$\|e(t)\| \leq C(t)\Delta t \ . \tag{8.12}$$

This is the first order accuracy of the forward Euler method.

It is important to note that this argument does not prove that the forward Euler method converges to the correct answer as $\Delta t \to 0$. Instead, it assumes the convergence and uses it to get a quantitative estimate of the error. The formal proof of convergence may be found in any good book on numerical methods for ODEs, such as the book by Isaiah Iserles.

If this analysis is done a little more carefully, it shows that there is an asymptotic error expansion

$$x_k \sim x(t_k) + \Delta t u_1(t_k) + \Delta t^2 u_2(t_k) + \cdots \ . \tag{8.13}$$

The leading error coefficient, $u_1(t)$, is the solution of (8.10). The higher order coefficients, $u_2(t)$, etc. are found by solving higher order error propagation equations.

The *modified equation* is a different approach to error analysis that allows us to determine the long time behavior of the error. The idea is to modify the differential equaion (8.1) so that the solution is closer to the forward Euler sequence. We know that $x_k = x(t_k) + O(\Delta t)$. We seek a differential equation $\dot{y} = g(y)$ so that $x_k = y(t_k) + O(\Delta t^2)$ We construct an error expansion for the equation itself rather than the solution.

It is simpler to require $y(t)$ so satisfy the forward Euler equation at each t, not just the discrete times t_k:

$$y(t + \Delta t) = y(t) + \Delta t f(y(t)) \ . \tag{8.14}$$

We seek

$$g(y, \Delta t) = g_0(y) + \Delta t g_1(y) + \cdots \tag{8.15}$$

so that the solution of (8.14) satisfies $\dot{y} = g(y) + O(\Delta t^2)$. We combine the expansion (8.15) with the Taylor series

$$y(t + \Delta t) = y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + O(\Delta t^3) \ ,$$

to get (dividing both sides by Δt, $O(\Delta t^3)/\Delta t = O(\Delta t^2)$.):

$$\begin{aligned} y(t) + \Delta t \dot{y}(t) + \frac{\Delta t^2}{2}\ddot{y}(t) + O(\Delta t^3) &= y(t) + \Delta t f(y(t)) \\ g_0(y(t)) + \Delta t g_1(y(t)) + \frac{\Delta}{2}\ddot{y}(t) &= f(y(t)) + O(\Delta t^2) \end{aligned}$$

[4]The value of $e(t)$ depends on the values of $r(s)$ for $0 \leq s \leq t$. We can solve $\dot{u} = f'(x)u - w$, where $w = \frac{1}{2}\ddot{x}$, then solve (8.10) by setting $e = \Delta t u$. This shows that $\|e(t)\| = \Delta t \, \|u(t)\|$, which is what we want, with $C(t) = \|u(t)\|$.

Equating the leading order terms gives the unsurprising result

$$g_0(y) = f(y) \ ,$$

and leaves us with

$$g_1(y(t)) + \frac{1}{2}\ddot{y}(t) = O(\Delta t) \ . \tag{8.16}$$

We differentiate $\dot{y} = f(y) + O(\Delta t)$ and use the chain rule, giving

$$\begin{aligned} \ddot{y} = \frac{d}{dt}\dot{y} &= \frac{d}{dt}\Big(f(y(t)) + O(\Delta t)\Big) \\ &= f'(y)\dot{y}(t) + O(\Delta t) \\ \ddot{y} &= f'(y)f(y) + O(\Delta t) \end{aligned}$$

Substituting this into (8.16) gives

$$g_1(y) = -\frac{1}{2}f'(y)f(y) \ .$$

so the modified equation, with the first correction term, is

$$\dot{y} = f(y) - \frac{\Delta t}{2}f'(y)f(y) \ . \tag{8.17}$$

A simple example illustrates these points. The nondimensional harmonic oscillator equation is $\ddot{r} = -r$. The solution is $r(t) = a\sin(t) + b\cos(t)$, which oscillates but does not grow or decay. We write this in first order as $\dot{x}_1 = x_2$, $\dot{x}_2 = -x_1$, or

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_2 \\ -x_1 \end{pmatrix} \ . \tag{8.18}$$

Therefore, $f(x) = \begin{pmatrix} x_2 \\ -x_1 \end{pmatrix}$, $f' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$, and $f'f = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} x_2 \\ -x_1 \end{pmatrix} = \begin{pmatrix} -x_1 \\ -x_2 \end{pmatrix}$, so (8.17) becomes

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix} + \frac{\Delta t}{2}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \frac{\Delta t}{2}t & 1 \\ -1 & \frac{\Delta t}{2} \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \ .$$

We can solve this by finding eigenvalues and eigenvectors, but a simpler trick is to use a partial integrating factor and set $y(t) = e^{\frac{1}{2}\Delta t \cdot t}z(t)$, where $\dot{z} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} z$. Since $z_1(t) = a\sin(t) + b\cos(t)$, we have our approximate numerical solution $y_1(t) = e^{\frac{1}{2}\Delta t \cdot t}\left(a\sin(t) + b\cos(t)\right)$. Therefore

$$\|e(t)\| \approx \left(e^{\frac{1}{2}\Delta t \cdot t} - 1\right) \ . \tag{8.19}$$

This modified equation analysis confirms that forward Euler is first order accurate. For small Δt, we write $e^{\frac{1}{2}\Delta t \cdot t} - 1 \approx \frac{1}{2}\Delta t \cdot t$ so the error is about $\Delta t \cdot t\left(a\sin(t) + b\cos(t)\right)$. Moreover, it shows that the error grows with t. For each fixed t, the error satisfies $\|e(t)\| = O(\Delta t)$ but the implied constant $C(t)$ (in $\|e(t)\| \leq C(t)\Delta t$) is a growing function of t, at least as large as $C(t) \geq \frac{t}{2}$.

8.2 Runge Kutta methods

Runge Kutta[5] methods are a general way to achieve higher order approximate solutions of the initial value problem (8.1), (8.2). Each time step consists of m stages, each stage involving a single evaluation of f. The relatively simple four stage fourth order method is in wide use today. Like the forward Euler method, but unlike multistep methods, Runge Kutta time stepping computes x_{k+1} from x_k without using values x_j for $j < k$. This simplifies error estimation and adaptive time step selection.

The simplest Runge Kutta method is forward Euler (8.5). Among the second order methods is Heun's[6]

$$\begin{aligned} \xi_1 &= \Delta t f(x_k, t_k) && (8.20)\\ \xi_2 &= \Delta t f(x_k + \xi_1, t_k + \Delta t) && (8.21)\\ x_{k+1} &= x_k + \frac{1}{2}(\xi_1 + \xi_2) \ . && (8.22) \end{aligned}$$

The calculations of ξ_1 and ξ_2 are the two *stages* of Heun's method. Clearly they depend on k, though that is left out of the notation.

To calculate x_k from x_0 using a Runge Kutta method, we apply take k time steps. Each time step is a transformation that may be written

$$x_{k+1} = \widehat{S}(x_k, t_k, \Delta t) \ .$$

As in Chapter 6, we express the general time step as[7] $x' = \widehat{S}(\overline{x}, t, \Delta t)$. This $\widehat{S}$ approximates the exact *solution operator*, $S(\overline{x}, t, \Delta t)$. We say that $x' = S(\overline{x}, t, \Delta t)$ if there is a trajectory satisfying the differential equation (8.1) so that $x(t) = \overline{x}$ and $x' = x(t + \Delta t)$. In this notation, we would give Heun's method as $x' = \widehat{S}(\overline{x}, \Delta t) = \overline{x} + \frac{1}{2}(\xi_1 + \xi_2)$, where $\xi_1 = f(\overline{x}, t, \Delta t)$, and $\xi_2 = f(\overline{x} + \xi_1, t, \Delta t)$.

The best known and most used Runge Kutta method, which often is called *the* Runge Kutta method, has four stages and is fourth order accurate

$$\begin{aligned} \xi_1 &= \Delta t f(\overline{x}, t) && (8.23)\\ \xi_2 &= \Delta t f(\overline{x} + \frac{1}{2}\xi_1, t + \frac{1}{2}\Delta t) && (8.24)\\ \xi_3 &= \Delta t f(\overline{x} + \frac{1}{2}\xi_2, t + \frac{1}{2}\Delta t) && (8.25)\\ \xi_4 &= \Delta t f(\overline{x} + \xi_3, t + \Delta t) && (8.26)\\ x' &= \overline{x} + \frac{1}{6}(\xi_1 + 2\xi_2 + 2\xi_3 + \xi_4) \ . && (8.27) \end{aligned}$$

[5] Carl Runge was Professor of applied mathematics at the turn of the 20^{th} century in Göttingen, Germany. Among his colleagues were David Hilbert (of Hilbert space) and Richard Courant. But Courant was forced to leave Germany and came to New York to found the Courant Institute. Kutta was a student of Runge.

[6] Heun, whose name rhymes with "coin", was another student of Runge.

[7] The notation x' here does not mean the derivative of x with respect to t (or any other variable) as it does in some books on differential equations.

Understanding the accuracy of Runge Kutta methods comes down to Taylor series. The reasoning of Section 8.1 suggests that the method has error $O(\Delta t^p)$ if

$$\widehat{S}(\overline{x}, t, \Delta t) = S(\overline{x}, t, \Delta t) + \Delta t \; r \; , \tag{8.28}$$

where $\|r\| = O(\Delta t^p)$. The reader should verify that this definition of the residual, r, agrees with the definition in Section 8.1. The analysis consists of expanding both $S(\overline{x}, t, \Delta t)$ and $\widehat{S}(\overline{x}, t, \Delta t)$ in powers of Δt. If the terms agree up to order Δt^p but disagree at order Δt^{p+1}, then p is the order of accuracy of the overall method.

We do this for Heun's method, allowing f to depend on t as well as x. The calculations resemble the derivation of the modified equation (8.17). To make the expansion of S, we have $x(t) = \overline{x}$, so

$$x(t + \Delta t) = \overline{x} + \Delta t \dot{x}(t) + \frac{\Delta t^2}{2} \ddot{x}(t) + O(\Delta t^3) \; .$$

Differentiating with respect to t and using the chain rule gives:

$$\ddot{x} = \frac{d}{dt}\dot{x} = \frac{d}{dt} f(x(t), t) = f'(x(t), t)\dot{x}(t) + \partial_t f(x(t), t) \; ,$$

so

$$\ddot{x}(t) = f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \; .$$

This gives

$$S(\overline{x}, t, \Delta t) = \overline{x} + \Delta t f(\overline{x}, t) + \frac{\Delta t^2}{2} \left(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \right) + O(\Delta t^3) \; . \tag{8.29}$$

To make the expansion of $\widehat{S}$ for Heun's method, we first have $\xi_1 = \Delta t f(\overline{x}, t)$, which needs no expansion. Then

$$\begin{aligned} \xi_2 &= \Delta t f(\overline{x} + \xi_1, t + \Delta t) \\ &= \Delta t \left(f(\overline{x}, t) + f'(\overline{x}, t)\xi_1 + \partial_t f(\overline{x}, t)\Delta t + O(\Delta t^2) \right) \\ &= \Delta t f(\overline{x}, t) + \Delta t^2 \left(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \right) + O(\Delta t^3) \; . \end{aligned}$$

Finally, (8.22) gives

$$\begin{aligned} x' &= \overline{x} + \frac{1}{2} \left(\xi_1 + \xi_2 \right) \\ &= \overline{x} + \frac{1}{2} \Big\{ \Delta t f(\overline{x}, t) + \Big[\Delta t f(\overline{x}, t) + \Delta t^2 \left(f'(\overline{x}, t) f(\overline{x}, t) + \partial_t f(\overline{x}, t) \right) \Big] \Big\} + O(\Delta t^3) \end{aligned}$$

Comparing this to (8.29) shows that

$$\widehat{S}(\overline{x}, t, \Delta t) = S(\overline{x}, t, \Delta t) + O(\Delta t^3) \; .$$

which is the second order accuracy of Heun's method. The same kind of analysis shows that the four stage Runge Kutta method is fourth order accurate, but it would take a full time week. It was Kutta's thesis.

8.3 Linear systems and stiff equations

A good way to learn about the behavior of a numerical method is to ask what it would do on a properly chosen *model problem*. In particular, we can ask how an initial value problem solver would perform on a linear system of differential equations

$$\dot{x} = Ax \ . \tag{8.30}$$

We can do this using the eigenvalues and eigenvectors of A if the eigenvectors are not too ill conditioned. If[8] $Ar_\alpha = \lambda_\alpha r_\alpha$ and $x(t) = \sum_{\alpha=1}^{n} u_\alpha(t) r_\alpha$, then the components u_α satisfy the scalar differential equations

$$\dot{u}_\alpha = \lambda_\alpha u_\alpha \ . \tag{8.31}$$

Suppose $x_k \approx x(t_k)$ is the approximate solution at time t_k. Write $x_k = \sum_{\alpha=1}^{n} u_{\alpha k} r_\alpha$. For a majority of methods, including Runge Kutta methods and linear multistep methods, the $u_{\alpha k}$ (as functions of k) are what you would get by applying the same time step approximation to the scalar equation (8.31). The eigenvector matrix, R, (see Section **??**), that diagonalizes the differential equation (8.30) also diagonalizes the computational method. The reader should check that this is true of the Runge Kutta methods of Section 8.2.

One question this answers, at least for linear equations (8.30), is how small the time step should be. From (8.31) we see that the λ_α have units of 1/time, so the $1/|\lambda_\alpha|$ have units of time and may be called *time scales*. Since Δt has units of time, it does not make sense to say that Δt is small in an absolute sense, but only relative to other time scales in the problem. This leads to the following:
Possibility: *A time stepping approximation to (8.30) will be accurate only if*

$$\max_\alpha \Delta t \, |\lambda_\alpha| \ll 1 \ . \tag{8.32}$$

Although this *possibility* is not true in every case, it is a dominant technical consideration in most practical computations involving differential equations. The *possibility* suggests that the time step should be considerably smaller than the smallest time scale in the problem, which is to say that Δt should *resolve* even the fastest time scales in the problem.

A problem is called *stiff* if it has two characteristics: (*i*) there is a wide range of time scales, and (*ii*) the fastest time scale modes have almost no energy. The second condition states that if $|\lambda_\alpha|$ is large (relative to other eigenvalues), then $|u_\alpha|$ is small. Most time stepping problems for partial differential equations are stiff in this sense. For a stiff problem, we would like to take larger time steps than (8.32):

$$\Delta t \, |\lambda_\alpha| \ll 1 \qquad \begin{cases} \text{for all } \alpha \text{ with } u_\alpha \text{ signifi-} \\ \text{cantly different from zero.} \end{cases} \tag{8.33}$$

What can go wrong if we ignore (8.32) and choose a time step using (8.33) is *numerical instability*. If mode u_α is one of the large $|\lambda_\alpha|$ small $|u_\alpha|$ modes,

[8] We call the eigenvalue index α to avoid conflict with k, which we use to denote the time step.

it is natural to assume that the real part satisfies $\mathrm{Re}(\lambda_\alpha) \leq 0$. In this case we say the mode is *stable* because $|u_\alpha(t)| = |u_\alpha(0)|\, e^{\lambda_\alpha t}$ does not increase as t increases. However, if $\Delta t \lambda_\alpha$ is not small, it can happen that the time step approximation to (8.31) is unstable. This can cause the $u_{\alpha k}$ to grow exponentially while the actual u_α decays or does not grow. Exercise 8 illustrates this. Time step restrictions arising from stability are called *CFL* conditions because the first systematic discussion of this possibility in the numerical solution of partial differential equations was given in 1929 by Richard Courant, Kurt Friedrichs, and Hans Levy.

8.4 Adaptive methods

Adaptive means that the computational steps are not fixed in advance but are determined as the computation proceeds. Section 3.6, discussed an integration algorithm that chooses the number of integration points adaptively to achieve a specified overall accuracy. More sophisticated adaptive strategies choose the distribution of points to maximize the accuracy from a given amount of work. For example, suppose we want an $\widehat{I}$ for $I = \int_0^2 f(x)dx$ so that $\left|\widehat{I} - I\right| < .06$. It might be that we can calculate $I_1 = \int_0^1 f(x)dx$ to within .03 using $\Delta x = .1$ (10 points), but that calculating $I_2 = \int_1^2 f(x)dx$ to within .03 takes $\Delta x = .02$ (50 points). It would be better to use $\Delta x = .1$ for I_1 and $\Delta x = .02$ for I_2 (60 points total) rather than using $\Delta x = .02$ for all of I (100 points).

Adaptive methods can use *local error estimates* to concentrate computational resources where they are most needed. If we are solving a differential equation to compute $x(t)$, we can use a smaller time step in regions where x has large accelleration. There is an active community of researchers developing systematic ways to choose the time steps in a way that is close to optimal without having the overhead in choosing the time step become larger than the work in solving the problem. In many cases they conclude, and simple model problems show, that a good strategy is to *equidistribute* the *local truncation error*. That is, to choose time steps Δt_k so that the the local truncation error $\rho_k = \Delta t_k r_k$ is nearly constant.

If we have a variable time step Δt_k, then the times $t_{k+1} = t_k + \Delta t_k$ form an irregular *adapted mesh* (or adapted *grid*). Informally, we want to choose a mesh that *resolves* the solution, $x(t)$ being calculated. This means that knowing the $x_k \approx x(t_k)$ allows you make an accurate reconstruction of the function $x(t)$, say, by interpolation. If the points t_k are too far apart then the solution is *underresolved*. If the t_k are so close that $x(t_k)$ is predicted accurately by a few neighboring values ($x(t_j)$ for $j = k \pm 1, k \pm 2$, etc.) then $x(t)$ is *overresolved*, we have computed it accurately but paid too high a price. An efficient adaptive mesh avoids both underresolution and overresolution.

Figure 8.1 illustrates an adapted mesh with equidistributed interpolation error. The top graph shows a curve we want to resolve and a set of points that concentrates where the curvature is high. Also also shown is the piecewise linear

curve that connects the interpolation points. On the graph is looks as though the piecewise linear graph is closer to the curve near the center than in the smoother regions at the ends, but the error graph in the lower frame shows this is not so. The reason probably is that what is plotted in the bottom frame is the vertical distance between the two curves, while what we see in the picture is the two dimensional distance, which is less if the curves are steep. The bottom frame illustrates equidistribution of errer. The interpolation error is zero at the grid points and gets to be as large as about -6.3×10^{-3} in each interpolation interval. If the points were uniformly spaced, the interpolation error would be smaller near the ends and larger in the center. If the points were bunched even more than they are here, the interpolation error would be smaller in the center than near the ends. We would not expect such perfect equidistribution in real problems, but we might have errors of the same order of magnitude everywhere.

For a Runge Kutta method, the local truncation error is $\rho(x,t,\Delta t) = \widehat{S}(x,t,\Delta t) - S(x,t,\Delta t)$. We want a way to estimate ρ and to choose Δt so that $|\rho| = e$, where e is the value of the equidistributed local truncation error. We suggest a method related to Richardson extrapolation (see Section 3.3), comparing the result of one time step of size Δt to two time steps of size $\Delta t/2$. The best adaptive Runge Kutta differential equation solvers do not use this generic method, but instead use ingenious schemes such as the Runge Kutta Fehlberg five stage scheme that simultaneously gives a fifth order $\widehat{S}_5$, but also gives an estimate of the difference between a fourth order and a fifth order method, $\widehat{S}_5 - \widehat{S}_4$. The method described here does the same thing with eight function evaluations instead of five.

The Taylor series analysis of Runge Kutta methods indicates that $\rho(x,t,\Delta t) = \Delta t^{p+1}\sigma(x,t) + O(\Delta t^{p+2})$. We will treat σ as a constant because the all the x and t values we use are within $O(\Delta t)$ of each other, so variations in σ do not effect the principal error term we are estimating. With one time step, we get $x' = \widehat{S}(\overline{x},t,\Delta t)$ With two half size time steps we get first $\widetilde{x}_1 = \widehat{S}(\overline{x},t,\Delta t/2)$, then $\widetilde{x}_2 = \widehat{S}(\widetilde{x}_1, t+\Delta t/2, \Delta t/2)$.

We will show, using the Richardson method of Section 3.3, that

$$x' - \widetilde{x}_2 = \left(1 - 2^{-p}\right)\rho(\overline{x},t,\Delta t) + O(\Delta t^{p+1}) \; . \tag{8.34}$$

We need to use the *semigroup* property of the solution operator: If we "run" the exact solution from $\overline{x}$ for time $\Delta t/2$, then run it from there for another time $\Delta t/2$, the result is the same as running it from $\overline{x}$ for time Δt. Letting x be the solution of (8.1) with $x(t) = \overline{x}$, the formula for this is

$$\begin{aligned} S(\overline{x},t,\Delta t) &= S(x(t+\Delta t/2), t+\Delta t/2, \Delta t/2) \\ &= S(\, S(\overline{x},t,\Delta t/2)\,, t+\Delta t/2, \Delta t/2) \; . \end{aligned}$$

We also need to know that $S(x,t,\Delta t) = x + O(\Delta t)$ is reflected in the Jacobian matrix S' (the matrix of first partials of S with respect to the x arguments with t and Δt fixed)[9]: $S'(x,t,\Delta t) = I + O(\Delta t)$.

[9]This fact is a consequence of the fact that S is twice differentiable as a function of all its arguments, which can be found in more theoretical books on differential equations. The Jacobian of $f(x) = x$ is $f'(x) = I$.

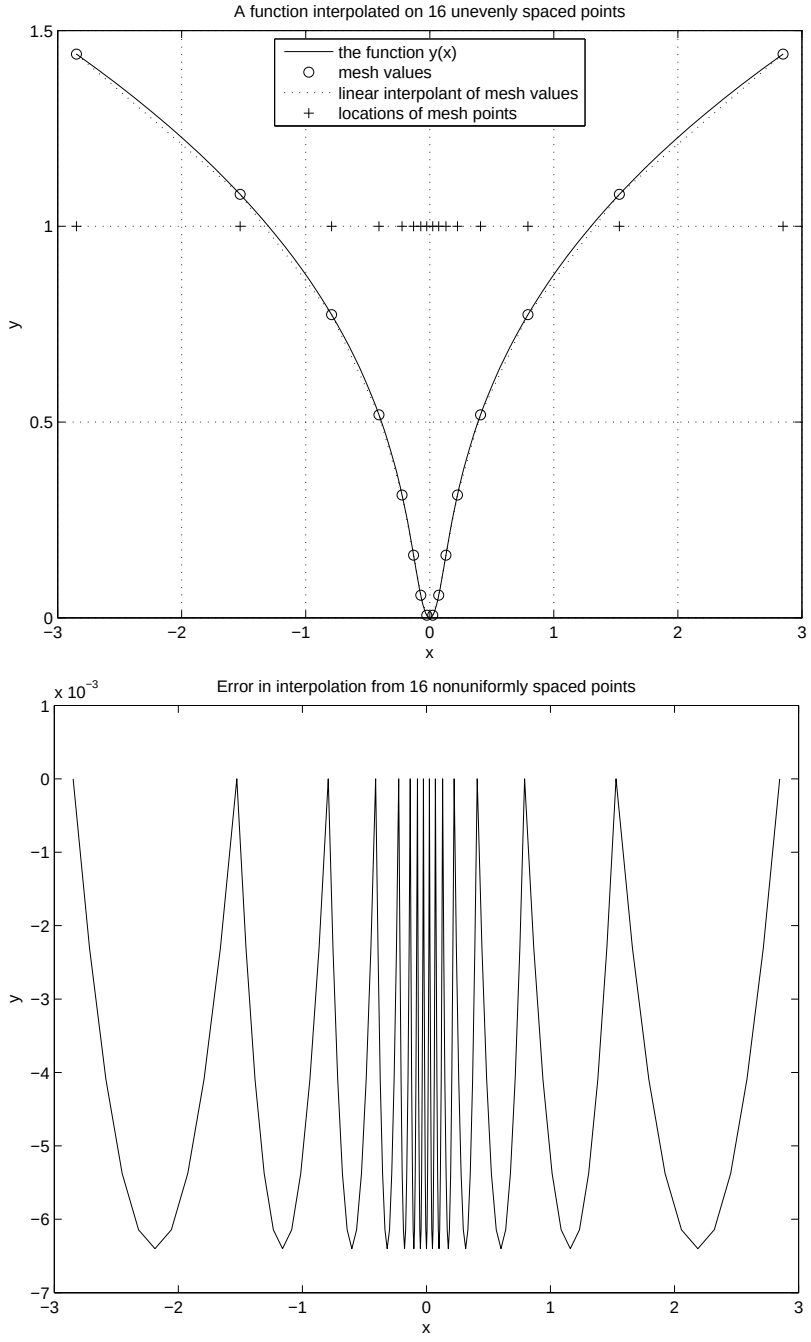

Figure 8.1: *A nonuniform mesh for a function that needs different resolution in different places. The top graph shows the function and the mesh used to interpolate it. The bottom graph is the difference between the function and the piecewise linear approximation. Note that the interpolation error equidistributed though the mesh is much finer near* $x = 0$.

The actual calculation starts with

$$\begin{aligned}\widetilde{x}_1 &= \widehat{S}(\overline{x}, t, \Delta t/2) \\ &= S(\overline{x}, t, \Delta t/2) + 2^{-(p+1)}\Delta t^{-(p+1)}\sigma + O(\Delta t^{-(p+2)}) \ ,\end{aligned}$$

and

$$\begin{aligned}\widetilde{x}_2 &= \widehat{S}(\widetilde{x}_1, t + \Delta t, \Delta t/2) \\ &= S(\widetilde{x}_1, t + \Delta t/2, \Delta t/2) + 2^{-(p+1)}\Delta t^{-(p+1)}\sigma + O(\Delta t^{-(p+2)}) \ ,\end{aligned}$$

We simplify the notation by writing $\widetilde{x}_1 = x(t+\Delta t/2)+u$ with $u = 2^{-(p+1)}\Delta t^p\sigma + O(\Delta t^{-(p+2)})$. Then $\|u\| = O(\Delta t^{-(p+1)})$ and also (used below) $\Delta t\,\|u\| = O(\Delta t^{-(p+2)})$ and (since $p \geq 1$) $\|u\|^2 = O(\Delta t^{-(2p+2)}) = O(\Delta t^{-(p+2)})$. Then

$$\begin{aligned}S(\widetilde{x}_1, t + \Delta t/2, \Delta t/2) &= S(x(t + \Delta t/2) + u, t + \Delta t/2, \Delta t/2) \\ &= S(x(t + \Delta t/2), t + \Delta t/2, \Delta t/2) + S'u + O\left(\|u\|^2\right) \\ &= S(x(t + \Delta t/2), t + \Delta t/2, \Delta t/2) + u + O\left(\|u\|^2\right) \\ &= S(\overline{x}, t, \Delta t) + 2^{-(p+1)}\Delta t^p\sigma + uO\left(\Delta t^{p+2}\right) \ .\end{aligned}$$

Altogether, since $2 \cdot 2^{-(p+1)} = 2^{-p}$, this gives

$$\widetilde{x}_2 = S(\overline{x}, t, \Delta t) + 2^{-p}\Delta t^{p+1}\sigma + O\left(\Delta t^{p+2}\right) \ .$$

Finally, a single size Δt time step has

$$x' = X(\overline{x}, \Delta t, t) + \Delta t^{p+1}\sigma + O\left(\Delta t^{p+2}\right) \ .$$

Combining these gives (8.34). It may seem like a mess but it has a simple underpinning. The whole step produces an error of order Δt^{p+1}. Each half step produces an error smaller by a factor of 2^{p+1}, which is the main idea of Richardson extrapolation. Two half steps produce almost exactly twice the error of one half step.

There is a simple adaptive strategy based on the local truncation error estimate (8.34). We arrive at the start of time step k with an estimated time step size Δt_k. Using that time step, we compute $x' = \widehat{S}(x_k, t_k, \Delta t_k)$ and $\widetilde{x}_2$ by taking two time steps from x_k with $\Delta t_k/2$. We then estimate ρ_k using (8.34):

$$\widehat{\rho}_k = \frac{1}{1 - 2^{-p}}\left(x' - \widetilde{x}_2\right) \ . \tag{8.35}$$

This suggests that if we adjust Δt_k by a factor of μ (taking a time step of size $\mu\Delta t_k$ instead of Δt_k), the error would have been $\mu^{p+1}\widehat{\rho}_k$. If we choose μ to exactly equidistribute the error (according to our estimated ρ, we would get

$$e = \mu^{p+1}\,\|\widehat{\rho}_k\| \implies \mu_k = \left(e/\,\|\widehat{\rho}_k\|\right)^{1/(p+1)} \ . \tag{8.36}$$

We could use this estimate to adjust Δt_k and calculate again, but this may lead to an infinite loop. Instead, we use $\Delta t_{k+1} = \mu_k \Delta t_k$.

Chapter 3 already mentioned the paradox of error estimation. Once we have a quantitative error estimate, we should use it to make the solution more accurate. This means taking

$$x_{k+1} = \widehat{S}(x_k, t_k, \Delta t_k) + \widehat{\rho}_k \ ,$$

which has order of accuracy $p+1$, instead of the order p time step $\widehat{S}$. This increases the accuracy but leaves you without an error estimate. This gives an order $p+1$ calculation with a mesh chosen to be nearly optimal for an order p calculation. Maybe the reader can find a way around this paradox. Some adaptive strategies reduce the overhead of error estimation by using the Richardson based time step adjustment, say, every fifth step.

One practical problem with this strategy is that we do not know the quantitative relationship between local truncation error and global error[10]. Therefore it is hard to know what e to give to achieve a given global error. One way to estimate global error would be to use a given e and get some time steps Δt_k, then redo the computation with each interval $[t_k, t_{k+1}]$ cut in half, taking exactly twice the number of time steps. If the method has order of accuracy p, then the global error should decrease very nearly by a factor of 2^p, which allows us to estimate that error. This is rarely done in practice. Another issue is that there can be isolated zeros of the leading order truncation error. This might happen, for example, if the local truncation error were proportional to a scalar function $\ddot{x}$. In (8.36), this could lead to an unrealistically large time step. One might avoid that, say, by replacing μ_k with $\min(\mu_k, 2)$, which would allow the time step to grow quickly, but not too much in one step. This is less of a problem for systems of equations.

8.5 Multistep methods

Linear multistep methods are the other class of methods in wide use. Rather than giving a general discussion, we focus on the two most popular kinds, methods based on difference approximations, and methods based on integrating $f(x(t))$, *Adams methods*. Hybrids of these are possible but often are unstable. For some reason, almost nothing is known about methods that both are multistage and multistep.

Multistep methods are characterized by using information from previous time steps to go from x_k to x_{k+1}. We describe them for a fixed Δt. A simple example would be to use the second order centered difference approximation $\dot{x}(t) \approx \left(x(t+\Delta t) - x(t-\Delta t) \right) / 2\Delta t$ to get

$$\left(x_{k+1} - x_{k-1} \right) / 2\Delta t = f(x_k) \ ,$$

[10] *Adjoint* based error control methods that address this problem are still in the research stage (2006).

or

$$x_{k+1} = x_{k-1} + 2\Delta t f(x_k) \ . \tag{8.37}$$

This is the *leapfrog*[11] method. We find that

$$\widetilde{x}_{k+1} = \widetilde{x}_{k-1} + 2\Delta t f(\widetilde{x}_k) + \Delta t O(\Delta t^2) \ ,$$

so it is second order accurate. It achieves second order accuracy with a single evaluation of f per time step. Runge Kutta methods need at least two evaluations per time step to be second order. Leapfrog uses x_{k-1} and x_k to compute x_{k+1}, while Runge Kutta methods forget x_{k-1} when computing x_{k+1} from x_k.

The next method of this class illustrates the subtletlies of multistep methods. It is based on the four point one sided difference approximation

$$\dot{x}(t) = \frac{1}{\Delta t}\left(\frac{1}{3}x(t+\Delta t) + \frac{1}{2}x(t) - x(t-\Delta t) + \frac{1}{6}x(t-2\Delta t)\right) + O\left(\Delta t^3\right) \ .$$

This suggests the time stepping method

$$f(x_k) = \frac{1}{\Delta t}\left(\frac{1}{3}x_{k+1} + \frac{1}{2}x_k - x_{k-1} + \frac{1}{6}x_{k-2}\right) \ , \tag{8.38}$$

which leads to

$$x_{k+1} = 3\Delta t f(x_k) - \frac{3}{2}x_k + 3x_{k-1} - \frac{1}{2}x_{k-2} \ . \tag{8.39}$$

This method never is used in practice because it is *unstable* in a way that Runge Kutta methods cannot be. If we set $f \equiv 0$ (to solve the model problem $\dot{x} = 0$), (8.38) becomes the *recurrence relation*

$$x_{k+1} + \frac{3}{2}x_k - 3x_{k-1} + \frac{1}{2}x_{k-2} = 0 \ , \tag{8.40}$$

which has *characteristic polynomial*[12] $p(z) = z^3 + \frac{3}{2}z^2 - 3z + \frac{1}{2}$. Since one of the roots of this polynomial has $|z| > 1$, general solutions of (8.40) grow exponentially on a Δt time scale, which generally prevents approximate solutions from converging as $\Delta t \to 0$. This cannot happen for Runge Kutta methods because setting $f \equiv 0$ always gives $x_{k+1} = x_k$, which is the exact answer in this case.

Adams methods use old values of f but not old values of x. We can integrate (8.1) to get

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} f(x(t))dt \ . \tag{8.41}$$

An accurate estimate of the integral on the right leads to an accurate time step. *Adams Bashforth* methods estimate the integral using polynomial extrapolation

[11] Leapfrog is a game in which two or more children move forward in a line by taking turns jumping over each other, as (8.37) jumps from x_{k-1} to x_{k+1} using only $f(x_k)$.

[12] If $p(z) = 0$ then $x_k = z^k$ is a solution of (8.40).

from earlier f values. At its very simplest we could use $f(x(t)) \approx f(x(t_k))$, which gives

$$\int_{t_k}^{t_{k+1}} f(x(t))dt \approx (t_{k+1} - t_k) f(x(t_k)) \ .$$

Using this approximation on the right side of (8.41) gives forward Euler.

The next order comes from linear rather than constant extrapolation:

$$f(x(t)) \approx f(x(t_k)) + (t - t_k) \frac{f(x(t_k)) - f(x(t_{k-1}))}{t_k - t_{k-1}} \ .$$

With this, the integral is estimated as (the generalization to non constant Δt is Exercise **??**):

$$\begin{aligned} \int_{t_k}^{t_{k+1}} f(x(t))dt &\approx \Delta t f(x(t_k)) + \frac{\Delta t^2}{2} \frac{f(x(t_k)) - f(x(t_{k-1}))}{\Delta t} \\ &= \Delta t \left[\frac{3}{2} f(x(t_k)) - \frac{1}{2} f(x(t_{k-1})) \right] \ . \end{aligned}$$

The second order Adams Bashforth method for constant Δt is

$$x_{k+1} = x_k + \Delta t \left[\frac{3}{2} f(x_k) - \frac{1}{2} f(x_{k-1}) \right] \ . \tag{8.42}$$

To program higher order Adams Bashforth methods we need to evaluate the integral of the interpolating polynomial. The techniques of polynomial interpolation from Chapter **??** make this simpler.

Adams Bashforth methods are attractive for high accuracy computations where stiffness is not an issue. They cannot be unstable in the way (8.39) is because setting $f \equiv 0$ results (in (8.42), for example) in $x_{k+1} = x_k$, as for Runge Kutta methods. Adams Bashforth methods of any order or accuracy require one evaluation of f per time step, as opposed to four per time step for the fourth order Runge Kutta method.

8.6 Implicit methods

Implicit methods use $f(x_{k+1})$ in the formula for x_{k+1}. They are used for stiff problems because they can be stable with large $\lambda \Delta t$ (see Section 8.3) in ways explicit methods, all the ones discussed up to now, cannot. An implicit method must solve a system of equations to compute x_{k+1}.

The simplest implicit method is *backward Euler*:

$$x_{k+1} = x_k + \Delta t f(x_{k+1}) \ . \tag{8.43}$$

This is only first order accurate, but it is stable for any λ and Δt if $\mathrm{Re}(\lambda) \leq 0$. This makes it suitable for solving stiff problems. It is called implicit because x_{k+1} is determined implicitly by (8.43), which we rewrite as

$$F(x_{k+1}, \Delta t) = 0 \quad , \quad \text{where } \ F(y, \Delta t) = y - \Delta t f(y) - x_k \ , \tag{8.44}$$

To find x_{k+1}, we have to solve this system of equations for y.

Applied to the linear scalar problem (8.31) (dropping the α index), the method (8.43) becomes $u_{k+1} = u_k + \Delta t \lambda u_{k+1}$, or

$$u_{k+1} = \frac{1}{1 - \Delta t \lambda} u_k \ .$$

This shows that $|u_{k+1}| < |u_k|$ if $\Delta t > 0$ and λ is any complex number with $\mathrm{Re}(\lambda) \leq 0$. This is in partial agreement with the qualitative behavior of the exact solution of (8.31), $u(t) = e^{\lambda t} u(0)$. The exact solution decreases in time if $\mathrm{Re}(\lambda) < 0$ but not if $\mathrm{Re}(\lambda) = 0$. The backward Euler approximation decreases in time even when $\mathrm{Re}(\lambda) = 0$. The backward Euler method artificially stabilizes a neutrally stable system, just as the forward Euler method artificially destabilizes it (see the modified equation discussion leading to (8.19)).

Most likely the equations (8.44) would be solved using an iterative method as discussed in Chapter 6. This leads to *inner iterations*, with the *outer* iteration being the time step. If we use the unsafeguarded local Newton method, and let j index the inner iteration, we get $F' = I - \Delta t f'$ and

$$y_{j+1} = y_j - \Big(I - \Delta t f'(y_j) \Big)^{-1} \big(y_j - \Delta t f(y_j) - x_k \big) \ , \tag{8.45}$$

hoping that $y_j \to x_{k+1}$ as $j \to \infty$. We can take initial guess $y_0 = x_k$, or, even better, an extrapolation such as $y_0 = x_k + \Delta t (x_k - x_{k-1}) / \Delta t = 2x_k - x_{k-1}$. With a good initial guess, just one Newton iteration should give x_{k+1} accurately enough.

Can we use the approximation $J \approx I$ for small Δt? If we could, the Newton iteration would become the simpler *functional iteration* (check this)

$$y_{j+1} = x_k + \Delta t f(y_j) \ . \tag{8.46}$$

The problem with this is that it does not work precisely for the stiff systems we use implicit methods for. For example, applied to $\dot{u} = \lambda u$, the functional iteration diverges ($|y_j| \to \infty$ as $j \to \infty$) for $\Delta t \lambda < -1$.

Most of the explicit methods above have implicit anologues. Among implicit Runge Kutta methods we mention the *trapezoid rule*

$$\frac{x_{k+1} - x_k}{\Delta t} = \frac{1}{2} \big(f(x_{k+1}) + f(x_k) \big) \ . \tag{8.47}$$

There are *backward differentiation formula*, or *BDF* methods based on higher order one sided approximations of $\dot{x}(t_{k+1})$. The second order BDF method uses (**??**):

$$\dot{x}(t) = \frac{1}{\Delta t} \left(\frac{3}{2} x(t) - 2x(t - \Delta t) + \frac{1}{2} x(t - 2\Delta t) \right) + O\left(\Delta t^2 \right) \ ,$$

to get

$$f(x(t_{k+1})) = \dot{x}(t_{k+1}) = \left(\frac{3}{2} x(t_{k+1}) - 2x(t_k) + \frac{1}{2} x(t_{k-1}) \right) + O\left(\Delta t^2 \right) \ ,$$

and, neglecting the $O\left(\Delta t^2\right)$ error term,

$$x_{k+1} - \frac{2\Delta t}{3} f(x_{k+1}) = \frac{4}{3} x_k - \frac{1}{3} x_{k-1} \ . \tag{8.48}$$

The *Adams Molton* methods estimate $\int_{t_k}^{t_{k+1}} f(x(t))dt$ using polynomial interpolation using the values $f(x_{k+1})$, $f(x_k)$, and possibly $f(x_{k-1})$, etc. The second order Adams Molton method uses $f(x_{k+1})$ and $f(x_k)$. It is the same as the trapezoid rule (8.47). The third order Adams Molton method also uses $f(x_{k-1})$. Both the trapezoid rule (8.47) and the second order BDF method (8.48) both have the property of being *A-stable*, which means being stable for (8.31) with any λ and Δt as long as $\mathrm{Re}(\lambda) \leq 0$. The higher order implicit methods are more stable than their explicit counterparts but are not A stable, which is a constant frustration to people looking for high order solutions to stiff systems.

8.7 Computing chaos, can it be done?

In many applications, the solutions to the differential equation (8.1) are *chaotic*.[13] The informal definition is that for large t (not so large in real applications) $x(t)$ is an unpredictable function of $x(0)$. In the terminology of Section 8.5, this means that the solution operator, $S(x_0, 0, t)$, is an ill conditioned function of x_0.

The dogma of Section **??** is that a floating point computation cannot give an accurate approximation to S if the condition number of S is larger than $1/\epsilon_{mach} \sim 10^{16}$. But practical calculations ranging from weather forcasting to molecular dynamics violate this rule routinely. In the computations below, the condition number of $S(x,t)$ increases with t and crosses 10^{16} by $t = 3$ (see Figure 8.3). Still, a calculation up to time $t = 60$ (Figure 8.4, bottom), shows the beautiful butterfly shaped *Lorentz attractor*, which looks just as it should.

On the other hand, in this and other computations, it truly is impossible to calculate details correctly. This is illustrated in Figure 8.2. The top picture plots two trajectories, one computed with $\Delta t = 4 \times 10^{-4}$ (dashed line), and the other with the time step reduced by a factor of 2 (solid line). The difference between the trajectories is an estimate of the accuracy of the computations. The computation seems somewhat accurate (curves close) up to time $t \approx 5$, at which time the dashed line goes up to $x \approx 15$ and the solid line goes down to $x \approx -15$. At least one of these is completely wrong. Beyond $t \approx 5$, the two "approximate" solutions have similar qualitative behavior but seem to be independent of each other. The bottom picture shows the same experiment with Δt a hundred times smaller than in the top picture. With a hundred times more accuracy, the approximate solution loses accuracy at $t \approx 10$ instead of $t \approx 5$. If a factor of 100 increase in accuracy only extends the validity of the solution by 5 time units, it should be hopeless to compute the solution out to $t = 60$.

[13] James Glick has written a nice popular book on chaos. Neil Strogatz has a more technical introduction that does not avoid the beautiful parts.

The present numerical experiments are on the *Lorentz equations*, which are a system of three nonlinear ordinary differential equations

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned}$$

with[14] $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$. The C/C++ program outputs (x, y, z) for plotting every $t = .02$ units of time, though there many time steps in each plotting interval. The solution first finds its way to the butterfly shaped *Lorentz attractor* then stays on it, travelling around the right and left wings in a seemingly random (technically, *chaotic*) way. The initial data $x = y = z = 0$ are not close to the attractor, so we ran the differential equations for some time before time $t = 0$ in order that $(x(0), y(0), z(0))$ should be a typical point on the attractor. Figure 8.2 shows the chaotic sequence of wing choice. A trip around the left wing corresponds to a dip of $x(t)$ down to $x \approx -15$ and a trip around the right wing corresponds to x going up to $x \approx 15$.

Section **??** explains that the condition number of the problem of calculating $S(x, t)$ (simplifying the notation from Section **??** to leave out zeros) depends on the Jacobian matrix $A(x, t) = \partial_x S(x, t)$. This represents the sensitivity of the solution at time t to small perturbations of the initial data. We can calculate $A(x, t)$ using ideas of perturbation theory similar to those we used for linear algebra problems in Chapter 4. Since $S(x, t)$ is the value of a solution at time t, it satisfies the differential equation

$$\frac{d}{dt} S(x, t) = f(S(x, t)) \ .$$

We differentiate both sides with respect to x and interchange the order of differentiation,

$$\frac{\partial}{\partial x} \frac{d}{dt} S(x, t) = \frac{d}{dt} \frac{\partial}{\partial x} S(x, t) = \frac{d}{dt} A(x, t) \ ,$$

to get (with the chain rule)

$$\begin{aligned} \frac{d}{dt} A(x, t) &= \frac{\partial}{\partial x} f(S(x, t)) \\ &= f'(S(x, t)) \cdot \partial_x S \\ \dot{A} &= f'(S(x, t)) A(x, t) \ . \end{aligned} \tag{8.49}$$

Thus, if we have an initial value x and calculate the trajectory $S(x, t)$, then we can calculate the *first variation*, $A(x, t)$, by solving the linear initial value problem (8.49) with initial condition $A(x, 0) = I$ (why?). In the present experiment, we solved the Lorentz equations and the perturbation equation using forward Euler with the same time step.

[14] These can be found, for example, in http://wikipedia.org by searching on "Lorentz attractor".

In typical chaotic problems, the first variation grows exponentially in time. If $\sigma_1(t) \geq \sigma_2(t) \geq \cdots \geq \sigma_n(t)$ are the singular values of $A(x,t)$, then there typically are *Lyapounov exponents*, μ_k, so that

$$\sigma_k(t) \sim e^{\mu_k t} \ ,$$

More precisely,

$$\lim_{t \to \infty} \frac{\ln(\sigma_k(t))}{t} = \mu_k \ .$$

If $\mu_1 > \mu_n$, the l^2 condition number of A grows exponentially,

$$\kappa_{l^2}(A(x,t)) = \frac{\sigma_1(t)}{\sigma_n(t)} \sim e^{(\mu_1 - \mu_n)t} \ .$$

Figure 8.3 gives some indication that our Lorentz system has differing Lyapoounov exponents. The top figure shows computations of the three singular values for $A(x,t)$. For $0 \leq t <\approx 2$, it seems that σ_3 is decaying exponentially, making a downward sloping straight line on this log plot. When σ_3 gets to about 10^{-15}, the decay halts. This is because it is nearly impossible for a full matrix in double precision floating point to have a condition number larger than $1/\epsilon_{mach} \approx 10^{16}$. When σ_3 hits 10^{-15}, we have $\sigma_1 \sim 10^2$, so this limit has been reached. These trends are clearer in the top frame of Figure 8.4, which is the same calculation carried to a larger time. Here $\sigma_1(t)$ seems to be growing exponentially with a gap between σ_1 and σ_3 of about $1/\epsilon_{mach}$. Theory says that σ_2 should be close to one, and the computations are consistent with this until the condition number bound makes $\sigma_2 \sim 1$ impossible.

To summarize, some results are quantitatively right, including the butterfly shape of the attractor and the exponential growth rate of $\sigma_1(t)$. Some results are qualitatively right but quantitatively wrong, including the values of $x(t)$ for $t >\approx 10$. Convergence analyses (comparing Δt results to $\Delta t/2$ results) distinguishes right from wrong in these cases. Other features of the computed solution are consistent over a range of Δt and consistently wrong. There is no reason to think the condition number of $A(x,t)$ grows exponentially until $t \sim 2$ then levels off at about 10^{16}. Much more sophisticated computational methods using the semigroup property show this is not so.

8.8 Software: Scientific visualization

Visualization of data is indispensable in scientific computing and computational science. Anomolies in data that seem to jump off the page in a plot are easy to overlook in numerical data. It can be easier to interpret data by looking at pictures than by examining columns of numbers. For example, here are entries 500 to 535 from the time series that made the top curve in the top frame of Figure 8.4 (multiplied by 10^{-5}).

```
0.1028    0.1020    0.1000    0.0963    0.0914    0.0864    0.0820
```

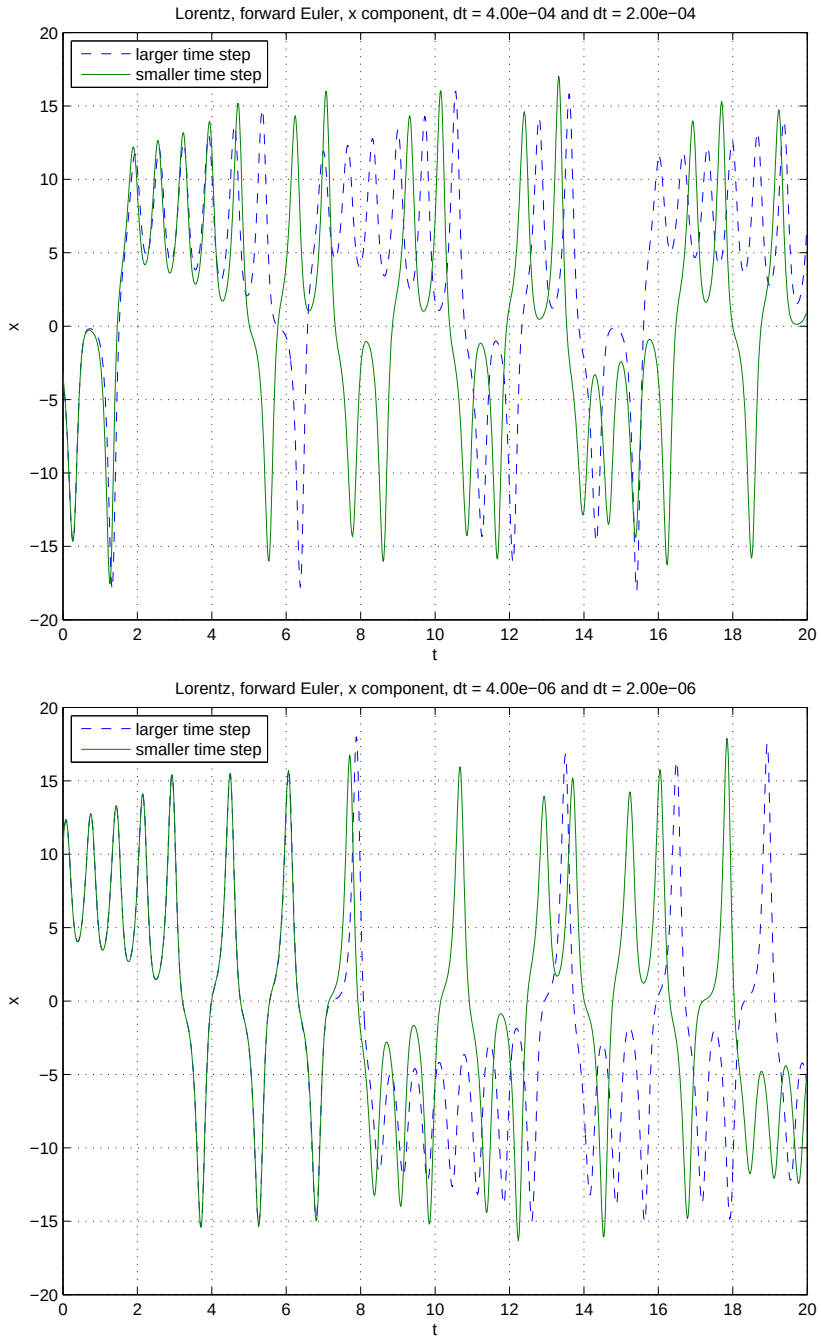

Figure 8.2: *Two convergence studies for the Lorentz system. The time steps in the bottom figure are 100 times smaller than the time steps in the to figure. The more accurate calculation loses accuracy at $t \approx 10$, as opposed to $t \approx 5$ with a larger time step. The qualitative behavior is similar in all computations.*

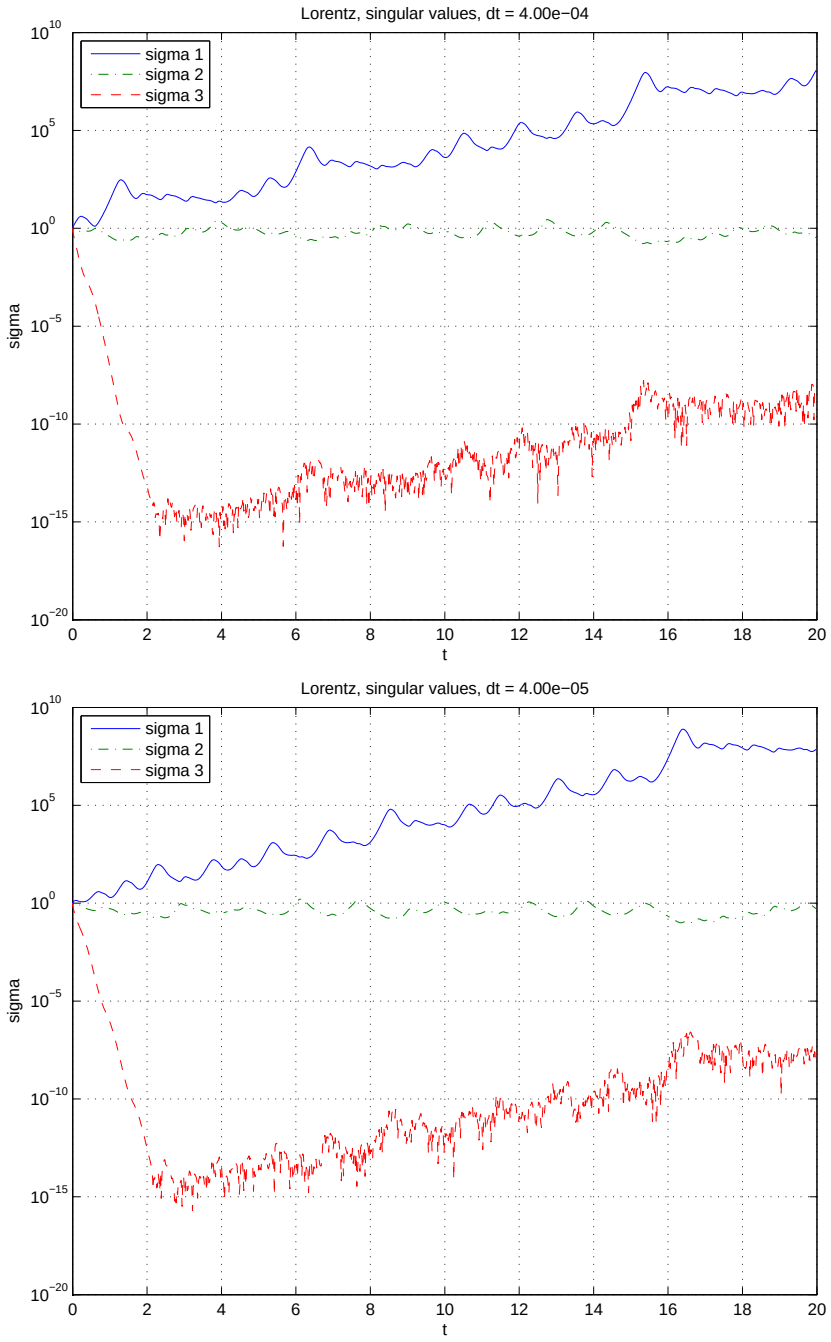

Figure 8.3: *Computed singluar values of the sensitivity matrix* $A(x,t) = \partial_x S(x,t)$ *with large time step (top) and ten times smaller time step (bottom). Top and bottom curves are similar qualitatively though the fine details differ. Theory predicts that middle singular value should be not grow or decay with* t*. The times from Figure 8.2 at which the numerical solution loses accuracy are not apparent here. In higher precision arithmetic,* $\sigma_3(t)$ *would have continued to decay exponentially. It is unlikely that computed singular values of any full matrix would differ by less than a factor of* $1/\epsilon_{mach} \approx 10^{16}$.

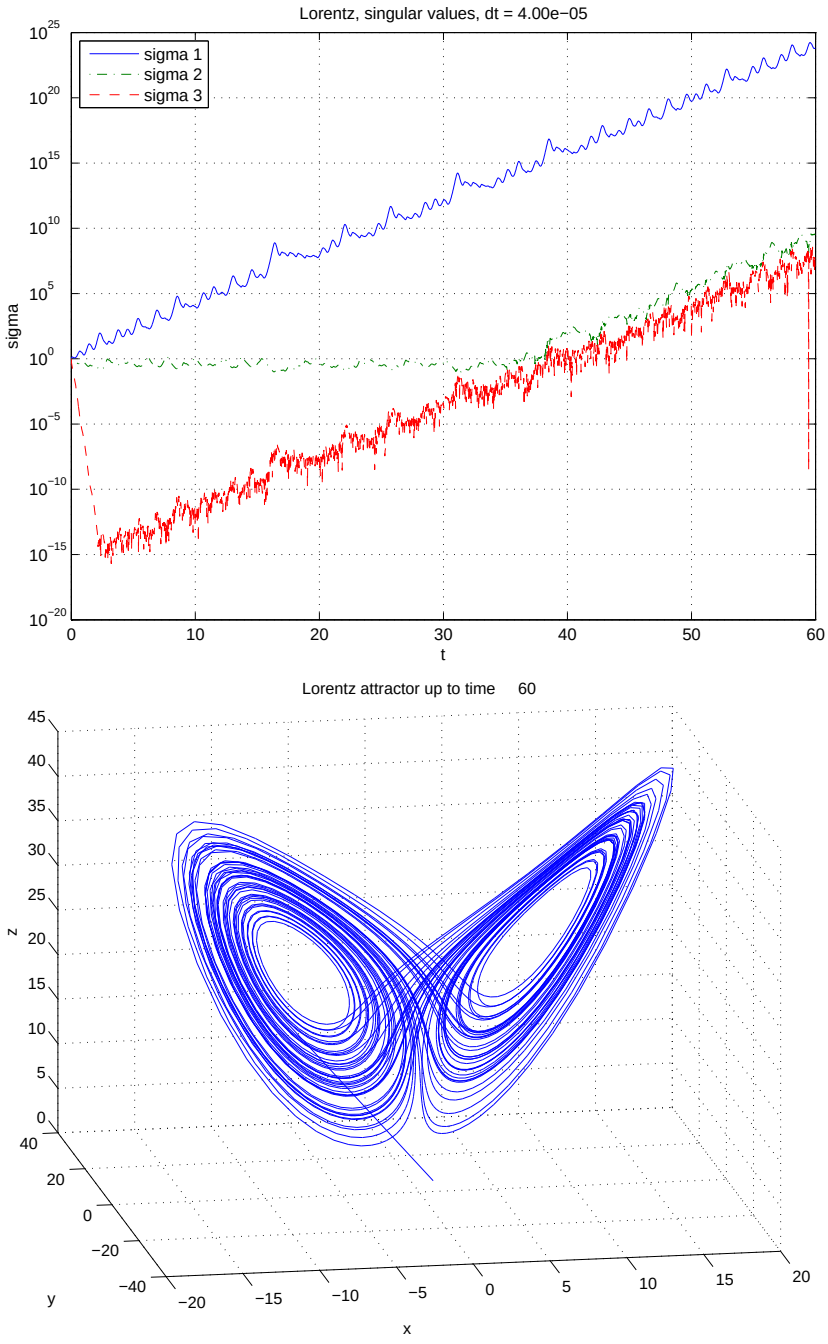

Figure 8.4: *Top: Singular values from Figure 8.3 computed for longer time. The $\sigma_1(t)$ grows exponentially, making a straight line on this log plot. The computed $\sigma_2(t)$ starts growing with the same exponential rate as σ_1 when roundoff takes over. A correct computation would show $\sigma_3(t)$ decreasing exponentially and $\sigma_2(t)$ neither growing nor decaying. Bottom: A beautiful picture of the butterfly shaped Lorentz attractor. It is just a three dimensional plot of the solution curve.*

```
0.0790    0.0775    0.0776    0.0790    0.0818    0.0857    0.0910
0.0978    0.1062    0.1165    0.1291    0.1443    0.1625    0.1844
0.2104    0.2414    0.2780    0.3213    0.3720    0.4313    0.4998
0.5778    0.6649    0.7595    0.8580    0.9542    1.0395    1.1034
```

Looking at the numbers, we get the overall impression that they are growing in an irregular way. The graph shows that the numbers have simple exponential growth with fine scale irregularities superposed. It could take hours to get that information directly from the numbers.

It can be a challenge to make visual sense of higher dimensional data. For example, we could make graphs of $x(t)$ (Figure 8.2), $y(t)$ and $z(t)$ as functions of t, but the single three dimensional plot in the lower frame of Figure 8.4 makes is clearer that the solution goes sometimes around the left wing and sometimes around the right. The three dimensional plot (`plot3` in Matlab) illuminates the structure of the solution better than three one dimensional plots.

There are several ways to visualize functions of two variables. A *contour plot* draws several *contour lines*, or *level lines*, of a function $u(x, y)$. A contour line for level u_k is the set of points (x, y) with $u(x, y) = u_k$. It is common to take about ten uniformly spaced values u_k, but most contour plot programs, including the Matlab program `contour`, allow the user to specify the u_k. Most contour lines are smooth curves or collections of curves. For example, if $u(x, y) = x^2 - y^2$, the contour line $u = u_k$ with $u_k \neq 0$ is a hyperbola with two components. An exception is $u_k = 0$, the contour line is an $\times$.

A *grid plot* represents a two dimensional rectangular array of numbers by colors. A *color map* assigns a color to each numerical value, that we call $c(u)$. In practice, usually we specify $c(u)$ by giving RGB values, $c(u) = (r(u), g(u), b(u))$, where r, g, and b are the intensities for red, green and blue respectively. These may be integers in a range (e.g. 0 to 255) or, as in Matlab, floating point numbers in the range from 0 to 1. Matlab uses the commands `colormap` and `image` to establish the color map and display the array of colors. The image is an array of boxes. Box (i, j) is filled with the color $c(u(i, j))$.

Surface plots visualize two dimensional surfaces in three dimensional space. The surface may be the graph of $u(x, y)$. The Matlab commands `surf` and `surfc` create surface plots of graphs. These look nice but often are harder to interpret than contour plots or grid plots. It also is possible to plot *contour surfaces* of a function of three variables. This is the set of (x, y, z) so that $u(x, y, z) = u_k$. Unfortunately, it is hard to plot more than one contour surface at a time.

Movies help visualize time dependent data. A movie is a sequence of frames, with each frame being one of the plots above. For example, we could visualize the Lorentz attractor with a movie that has the three dimensional butterfly together with a dot representing the position at time t.

The default in Matlab, and most other quality visualization packages, is to render the user's data as explicitly as possible. For example, the Matlab command `plot(u)` will create a piecewise linear "curve" that simply connects successive data points with straight lines. The plot will show the granularity of the data as well as the values. Similarly, the grid lines will be clearly visible in a

color grid plot. This is good most of the time. For example, the bottom frame of Figure 8.4 clearly shows the granularity of the data in the wing tips. Since the curve is sampled at uniform time increments, this shows that the trajectory is moving faster at the wing tips than near the body where the wings meet.

Some plot packages offer the user the option of smoothing the data using spline interpolation before plotting. This might make the picture less angular, but it can obscure features in the data and introduce artifacts, such as overshoots, not present in the actual data.

8.9 Resources and further reading

There is a beautiful discussion of computational methods for ordinary differential equations in *Numerical Methods* by øA]ke Bjöork and Germund Dahlquist. It was Dahlquist who created much of our modern understanding of the subject. A more recent book is *A First Course in Numerical Analysis of Differential Equations* by Arieh Iserles. The book *Numerical Solution of Ordinary Differential Equations* by Lawrence Shampine has a more practical orientation.

There is good public domain software for solving ordinary differential equations. A particularly good package is LSODE (google it).

The book by Sans-Serna explains symplectic integrators and their application to large scale problems such as the dynamics of large scape biological molecules. It is an active research area to understand the quantitative relationship between long time simulations of such systems and the long time behavior of the systems themselves. Andrew Stuart has written some thoughtful oapers on the subject.

The numerical solution of partial differential equations is a vast subject with many deep specialized methods for different kinds of problems. For computing stresses in structures, the current method of choice seems to be *finite element* methods. For fluid flow and wave propagation (particularly nonlinear), the majority relies on finite difference and finite volume methods. For finite differences, the old book by Richtmeyer and Morton still merit though there are more up to date books by Randy LeVeque and by Bertil Gustavson, Heinz Kreiss, and Joe Oliger.

8.10 Exercises

1. We compute the second error correction $u_2(t)$ n (8.13). For simplicity, consider only the scalar equation ($n = 1$). Assuming the error expansion, we have

$$\begin{aligned} f(x_k) &= f(\widetilde{x}_k + \Delta t u_1(t_k) + \Delta t^2 u_2(t_k) + O(\Delta t^3)) \\ &\approx f(\widetilde{x}_k) + f'(\widetilde{x}_k)\left(\Delta t u_1(t_k) + \Delta t^2 u_2(t_k)\right) \\ &\quad + \frac{1}{2} f''(\widetilde{x}_k)\Delta t^2 u_1(t_k)^2 + O\left(\Delta t^3\right) \ . \end{aligned}$$

Also

$$\frac{x(t_k+\Delta t)-x(t_k)}{\Delta t} = \dot{x}(t_k) + \frac{\Delta t}{2}\ddot{x}(t_k) + \frac{\Delta t^2}{6}x^{(3)}(t_k) + O\left(\Delta t^3\right) \ ,$$

and

$$\Delta t\frac{u_1(t_k+\Delta t)-u_1(t_k)}{\Delta t} = \Delta t\dot{u}_1(t_k) + \frac{\Delta t^2}{2}\ddot{u}_1(t_k) + O\left(\Delta t^3\right) \ .$$

Now plug in (8.13) on both sides of (8.5) and collect terms proportional to Δt^2 to get

$$\dot{u}_2 = f'(x(t))u_2 + \frac{1}{6}x^{(3)}(t) + \frac{1}{2}f''(x(t))u_1(t)^2 + ??? \ .$$

2. This exercise confirms what was hinted at in Section 8.1, that (8.19) correctly predicts error growth even for t so large that the solution has lost all accuracy. Suppose $k = R/\Delta t^2$, so that $t_k = R/\Delta t$. The error equation (8.19) predicts that the forward Euler approximation x_k has grown by a factor of $e^R/2$ although the exact solution has not grown at all. We can confirm this by direct calculation. Write the forward Euler approximation to (8.18) in the form $x_{k+1} = Ax_k$, where A is a 2×2 matrix that depends on Δt. Calculate the eigenvalues of A up to second order in Δt: $\lambda_1 = 1 + i\Delta t + a\Delta t^2 + O(\Delta t^3)$, and $\lambda_2 = 1 - i\Delta t + b\Delta t^2 + O(\Delta t^3)$. Find the constants a and b. Calculate $\mu_1 = \ln(\lambda_1) = i\Delta t + c\Delta t^2 + O(\Delta t^3)$ so that $\lambda_1 = \exp(i\Delta t + c\Delta t^2 + O(\Delta t^3))$. Conclude that for $k = R/\Delta t^2$, $\lambda_1^k = \exp(k\mu_1) = e^{iR/\Delta t}e^{R/2+O(\Delta t)}$, which shows that the solution has grown by a factor of nearly $e^{R/2}$ as predicted by (8.19). This s**t is good for something!

3. Another two stage second order Runge Kutta method sometimes is called the *modified Euler* method. The first stage uses forward Euler to predict the x value at the middle of the time step: $\xi_1 = \frac{\Delta t}{2}f(x_k, t_k)$ (so that $x(t_k + \Delta t/2) \approx x_k + \xi_1$). The second stage uses the midpoint rule with that estimate of $x(t_k + \Delta t/2)$ to step to time t_{k+1}: $x_{k+1} = x_k + \Delta t f(t_k + \frac{\Delta t}{2}, x_k + \xi_1)$. Show that this method is second order accurate.

4. Show that applying the four stage Runge Kutta method to the linear system (8.30) is equivalent to approximating the fundamental solution $S(\Delta t) = \exp(\Delta t A)$ by its Taylor series in Δt up to terms of order Δt^4 (see Exercise **??**). Use this to verify that it is fourth order for linear problems.

5. Write a C/C++ program that solves the initial value problem for (8.1), with f independent of t, using a constant time step, Δt. The arguments to the initial value problem solver should be T (the final time), Δt (the time step), $f(x)$ (specifying the differential equations), n (the number of components of x), and x_0 (the initial condition). The output should be the apaproximation to $x(T)$. The code must do something to preserve

the overall order of accuracy of the method in case T is not an integer multiple of Δt. The code should be able to switch between the three methods, forward Euler, second order Adams Bashforth, forth order four state Runge Kutta, with a minimum of code editing. Hand in output for each of the parts below showing that the code meets the specifications.

(a) The procedure should return an error flag or notify the calling routine in some way if the number of time steps called for is negative or impossibly large.

(b) For each of the three methods, verify that the coding is correct by testing that it gets the right answer, $x(.5) = 2$, for the scalar equation $\dot{x} = x^2$, $x(0) = 1$.

(c) For each of the three methods and the test problem of part b, do a convergence study to verify that the method achieves the theoretical order of accuracy. For this to work, it is necessary that T should be an integer multiple of Δt.

(d) Apply your code to problem (8.18) with initial data $x_0 = (1, 0)^*$. Repeat the convergence study with $T = 10$.

6. Verify that the recurrence relation (8.39) is unstable.

(a) Let z be a complex number. Show that the sequence $x_k = z^k$ satisfies (8.39) if and only if z satisfies $0 = p(z) = z^3 + \frac{3}{2}z^2 - 3z + \frac{1}{2}$.

(b) Show that $x_k = 1$ for all k is a solution of the recurrence relation. Conclude that $z = 1$ satisfies $p(1) = 0$. Verify that this is true.

(c) Using polynomial division (or another method) to factor out the known root $z = 1$ from $p(z)$. That is, find a quadratic polynomial, $q(z)$, so that $p(z) = (z - 1)q(z)$.

(d) Use the quadratic formula and a calculator to find the roots of q as $z = \frac{-5}{4} \pm \sqrt{\frac{41}{16}} \approx -2.85,\ .351$.

(e) Show that the general formula $x_k = az_1^k + bz_2^k + cz_3^k$ is a solution to (8.39) if z_1, z_2, and z_3 are three roots $z_1 = 1$, $z_2 \approx -2.85$, $z_3 \approx .351$, and, conversely, the general solution has this form. Hint: we can find a, b, c by solving a vanderMonde system (Section 7.4) using x_0, x_1, and x_2.

(f) Assume that $|x_0| \leq 1$, $|x_1| \leq 1$, and $|x_2| \leq 1$, and that b is on the order of double precision floating point roundoff (ϵ_{mach}) relative to a and c. Show that for $k > 80$, x_k is within ϵ_{mach} of bz_2^k. Conclude that for $k > 80$, the numerical solution has nothing in common with the actual solution $x(t)$.

7. Applying the implicit trapezoid rule (8.47) to the scalar model problem (8.31) results in $u_{k+1} = m(\lambda\Delta t)u_k$. Find the formula for m and show that $|m| \leq 1$ if $\mathrm{Re}(\lambda) \leq 0$, so that $|u_{k+1}| \leq |u_k|$. What does this say about the applicibility of the trapezoid rule to stiff problems?

8. Exercise violating time step constraint.

9. Write an adaptive code in C/C++ for the initial value problem (8.1) (8.2) using the method described in Section 8.4 and the four stage fourth order Runge Kutta method. The procedure that does the solving should have arguments describing the problem, as in Exercise 5, and also the local truncation error level, e. Apply the method to compute the trajectory of a comet. In nondimensionalized variables, the equations of motion are given by the inverse square law:

$$\frac{d^2}{dt^2}\begin{pmatrix} r_1 \\ r_2 \end{pmatrix} = \frac{-1}{(r_1^2 + r_2^2)^{3/2}}\begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \quad .$$

We always will suppose that the comet starts at $t = 0$ with $r_1 = 10$, $r_2 = 0$, $\dot{r}_1 = 0$, and $\dot{r}_2 = v_0$. If v_0 is not too large, the point $r(t)$ traces out an ellipse in the plane[15]. The shape of the ellipse depends on v_0. The *period*, $P(v_0)$, is the first time for which $r(P) = r(0)$ and $\dot{r}(P) = \dot{r}(0)$. The solution $r(t)$ is *periodic* because it has a period.

(a) Verify the correctness of this code by comparing the results to those from the fixed time step code from Exercise 5 with $T = 30$ and $v_0 = .2$.

(b) Use this program, with a small modification to compute $P(v_0)$ in the range $.01 \leq v_0 \leq .5$. You will need a criterion for telling when the comet has completed one orbit since it will not happen that $r(P) = r(0)$ exactly. Make sure your code tests for and reports failure[16].

(c) Choose a value of v_0 for which the orbit is rather but not extremely elliptical (width about ten times height). Choose a value of e for which the solution is rather but not extremely accurate – the error is small but shows up easily on a plot. If you set up your environment correctly, it should be quick and easy to find suitable paramaters by trial and error using Matlab graphics. Make a plot of a single period with two curves on one graph. One should be a solid line representing a highly accurate solution (so accurate that the error is smaller than the line width – *plotting accuracy*), and the other being the modestly accurate solution, plotted with a little "o" for each time step. Comment on the distribution of the time step points.

(d) For the same parameters as part b, make a single plot of that contains three curves, an accurate computation of $r_1(t)$ as a function of t (solid line), a modestly accurate computation of r_1 as a function of t ("o" for each time step), and Δt as a function of t. You will need to use a

[15] Isaac Newton formulated these equations and found the explicit solution. Many aspects of planetary motion – elliptical orbits, sun at one focus, $|r|\,\dot{\theta} = const$ – had beed discovered observationally by Johannes Keppler. Newton's inverse square law *theory* fit Keppler's *data*.

[16] This is not a drill.

different scale for Δt if for no other reason than that it has different units. Matlab can put one scale in the left and a different scale on the right. It may be necessary to plot Δt on a log scale if it varies over too wide a range.

(e) Determine the number of adaptive time stages it takes to compute $P(.01)$ to .1% accuracy (error one part in a thousand) and how many fixed Δt time step stages it takes to do the same. The counting will be easier if you do it within the function f.

10. The vibrations of a two dimensional crystal lattice may be modelled in a crude way using the differential equations[17]

$$\ddot{r}_{jk} = r_{j-1,k} + r_{j+1,k} + r_{j,k-1} + r_{j,k+1} - 4r_{jk} \ . \tag{8.50}$$

Here $r_{jk}(t)$ represents the displacement (in the vertical direction) of an atom at the (j,k) location of a square crystal lattice of atoms. Each atom is *bonded* to its four neighbors and is pulled toward them with a linear force. A lattice of size L has $1 \leq j \leq L$ and $1 \leq k \leq L$. Apply *reflecting* boundary conditions along the four boundaries. For example, the equation for $r_{1,k}$ should use $r_{0,k} = r_{1,k}$. This is easy to implement using a *ghost cell* strategy. Create ghost cells along the boundary and copy appropriate values from the actual cells to the ghost cells before each evaluation of f. This allows you to use the formula (8.50) at every point in the lattice. Start with initial data $r_{jk}(0) = 0$ and $\dot{r}_{jk}(0) = 0$ for all j,k except $\dot{r}_{1,1} = 1$. Compute for $L = 100$ and $T = 300$. Use the fourth order Runge Kutta method with a fixed time step $\Delta t = .01$. Write the solution to a file every .5 time units then use Matlab to make a movie of the results, with a 2D color plot of the solution in each frame. The movie should be a circular wave moving out from the bottom left corner and bouncing off the top and right boundaries. There should be some beautiful wave patterns inside the circle that will be hard to see far beyond time $t = 100$. Hand in a few of your favorite stills from the movie. If you have a web site, post your movie for others to enjoy.

[17] This is one of Einstein's contributions to science.