Section 1
ODE part 1, Runge Kutta methods

January 24, 2018

1 Motivation

This course, *Numerical Methods, II*, is about numerical methods for solving differential equations. It is natural to start with the the *initial value problem* for *ordinary differential equations*. This means computing a *trajectory* $x(t)$ that is determined by a system of differential equations

$$\frac{d}{dt}x = \dot{x} = f(x)$$

and *initial conditions*

$$x(t_0) = x_0 \ .$$

This chapter introduces *time stepping* methods for computing (approximate) trajectories. These methods approximate the trajectory by repeatedly advancing the solution by small increments of time Δt. The simplest example is *Euler's method*, which uses the approximation

$$x(t + \Delta t) \approx x(t) + \Delta t\, \dot{x}(t) = x(t) + \Delta t f(x(t)) \ .$$

This approximation is applied repeatedly to produce an approximate trajectory

$$\begin{aligned} x_1 &= x_0 + \Delta t f(x_0) \\ x_2 &= x_1 + \Delta t f(x_1) \\ &\text{etc.} \end{aligned}$$

These formulas produce approximations $x_n \approx x(t_n)$, where $t_n = t_0 + n\Delta t$. The Euler method is

$$x_{n+1} = x_n + \Delta t f(x_n) \ . \tag{1}$$

Two approximations are used to derive it. One is the Taylor approximation

$$x(t_{n+1}) \approx x(t_n) + \Delta t\, \dot{x}(t_n) = x(t_n) + \Delta t f(x_n) \ . \tag{2}$$

But $x(t_n)$ is not known, so we use the second approximation

$$x(t_n) \approx x_n \ . \tag{3}$$

You come to the Euler method (1) by substituting (3) into the right side of (2). This is taken to be the definition of the approximation x_{n+1}.

The *error* is the difference between the computed approximation and the actual solution to the differential equation.

$$E_n = x_n - x(t_n) \ .$$

The error depends on Δt. If the error goes to zero as $\Delta t \to 0$, we say the method *converges*.[1] If you try a computation with a given Δt and the error is

[1] Technically, it is possible to converge, in the sense of mathematical analysis, to the wrong answer as $\Delta t \to 0$. In numerical analysis, we usually mean *convergence* to mean convergence to the right answer.

too large, you can run your program with progressively smaller Δt until you get the accuracy you need.[2] Unfortunately, smaller Δt means you need more time steps to get to the same physical time. The total number of time steps is ($\lceil r \rceil$ is "ceiling" of r, which is the smallest integer $\geq r$)

$$N_T = \left\lceil \frac{T - t_0}{\Delta t} \right\rceil .$$

You double the work when you reduce Δt by a factor of 2.

In software, the Euler method would be embedded in an ODE solver package. The developer writes code to implement the generic formula (1). The user expresses the specific application by writing code to implement the function $f(x)$. The rest of the data about the problem also must be passed from the user to the developer code. This includes the dimension of the problem (number of components of x and f), the initial time, the initial condition x_0, and the desired final time T. Someone has to choose the time step size Δt, which is not as simple as it may seem.

The Python code `ODE_fE_Lorenz.py` illustrates this structure. It solves the 3D ODE system[3]

$$\left.\begin{array}{l} \dot{x}_0 = \sigma(x_1 - x_0) \\ \dot{x}_1 = x_0(\rho - x_2) - x_1 \\ \dot{x}_2 = x_0 x_1 - \beta x_2 \\ \sigma = 10 \ , \ \ \beta = \frac{8}{3} \ , \ \ \rho = 28 \ . \end{array}\right\} \tag{4}$$

from time $t_0 = 0$ to time $T = 10$. The specific ODE system is implemented in the code

```
f[0] = sig*(x[1]-x[0])              # evaluate f(x)
f[1] = x[0]*( rho - x[2]) - x[1]
f[2] = x[0]*x[1] - beta*x[2]
```

The Euler method is implemented in the code:

```
x = x + dt*f     # take a forward Euler time step
t = t + dt       # increment time, so x is x(t)
```

In this example, these two things (evaluate f, take a time step) are done within the same Python module (file) in the same code block. It is more common, and better programming practice, to separate the generic code from the problem specific code into separate modules. In Python, this is done using modules and sometimes the `class` mechanism.

[2] This sentence is naive. For one thing, it's hard to know how big the error is. For another, you probably don't have the computing resources to use a Δt as small as you want.

[3] There is a fascinating story behind this *Lorenz system*. Lorenz created it as a three equation model of something about the atmosphere (look it up). He was surprised to discover that roundoff level changes in initial conditions led to completely different trajectories. This was an important step in the modern appreciation of chaos.

Figure 1 plots the computed $x_2(t)$ for various values of Δt. For early times, the difference between $\Delta t = 2 \cdot 10^{-3}$ and $\Delta t = 10^{-3}$ (a factor of 2) is about the same as the difference between $\Delta t = 10^{-3}$ and $\Delta t = 10^{-4}$ (a factor of 10). The curves with $\Delta t = 10^{-4}$ and $\Delta t = 10^{-5}$ are almost indistinguishable (the $\Delta t = 10^{-4}$ curve covers the red $\Delta t = 10^{-5}$ curve, except that a little red is visible at the right end.). This indicates that the $\Delta t = 10^{-4}$ curve is correct to "plotting accuracy", meaning that the difference is less than the thickness of lines in the plot.

There are better ways to compute a generic ODE trajectory. The Python code `ODE_trap_Lorenz.py` is an implementation of the method:

$$k_1 = f(x_n) \tag{5}$$

$$k_2 = f(x_n + \tfrac{1}{2}\Delta t\, k_1) \tag{6}$$

$$x_{n+1} = x_n + \Delta t\, k_2 \tag{7}$$

Of course, k_1 and k_2 are different for each n. It would be more correct to write $k_{n,1}$ and $k_{n,2}$, but it is traditional to leave out the n subscript with the intermediate values k_*. One time step consists of the two *stages* (5) and (6). A stage is an evaluation of f. The new value x_{n+1} is a function of or the old value

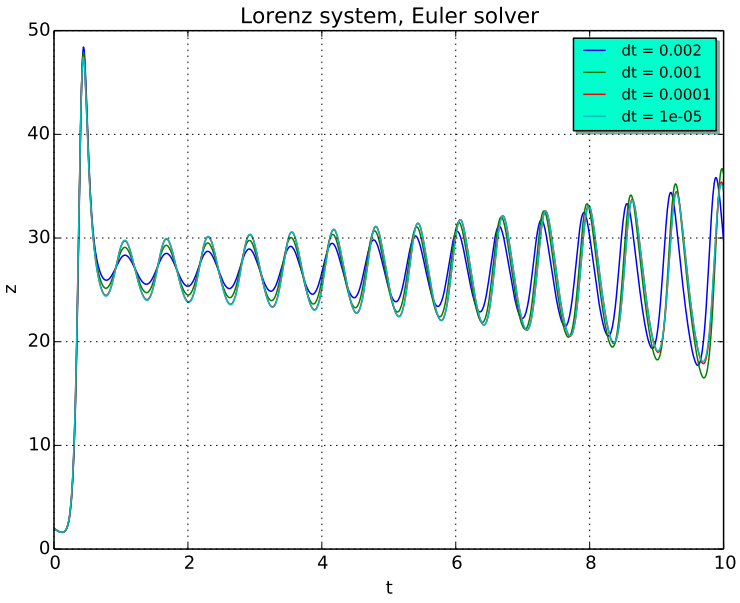

Figure 1: Forward Euler approximations to a the Lorenz system. A visual convergence study with several values of Δt. The curves for $\Delta t = 10^{-4}$ and $\Delta t = 10^{-5}$ are almost indistinguishable on this plot. The Python source is in `codes/ODE_fE.py`.

x_n, but a more complicated one than the Euler method. The function may be written explicitly as

$$x_{n+1} = x_n + \Delta t f(x_n + \tfrac{1}{2}\Delta t f(x_n)) \; . \tag{8}$$

A time step of this method is more expensive than for Euler. It uses two f evaluations instead of one, and it requires storage for the intermediate values k_1 and k_2.

Figure 2 shows that, in this example, the two stage method achieves plotting accuracy (the red $\Delta t = 10^{-3}$ line completely covers the $\Delta t = 2 \cdot 10^{-3}$ line) with $\Delta t = 2 \cdot 10^{-3}$. This time step is 20 times larger than $\Delta t = 10^{-4}$ that was needed with the Euler method. This means 20 times fewer time steps. Since there are two f evaluations per time step, that means 10 times fewer f evaluations. The same accuracy is achieved with about a tenth of the work.

Runge Kutta methods are time stepping methods that compute x_{n+1} as a function of x_n in some way. The other main class is *multistep* methods, which compute x_{n+1} using more information from the past, say x_n and x_{n-1}. We study them later. Runge Kutta methods are *one-step* methods.

Runge Kutta methods are derived using direct but laborious Taylor series

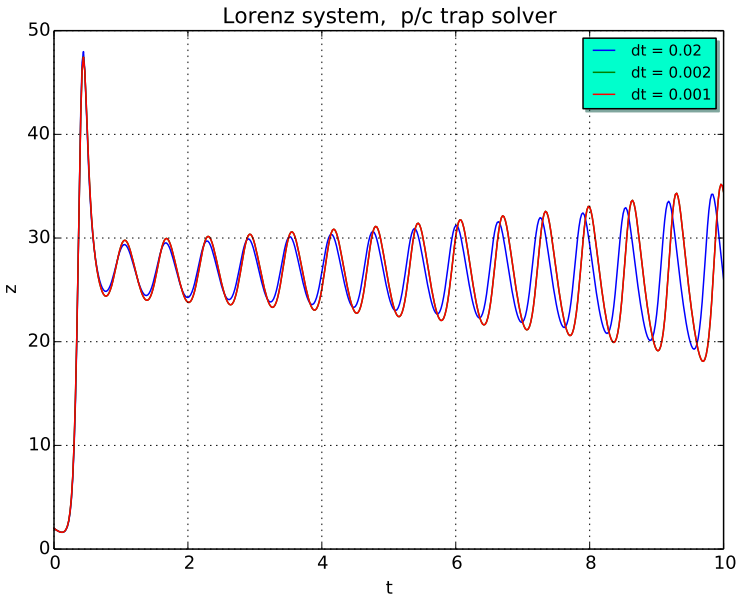

Figure 2: The ODE system of Figure 1 solved by a two stage method. This method achieves plotting accuracy with $\Delta t = 2 \cdot 10^{-3}$, which is 20 times larger than was needed in Figure 1. The Python source is in `codes/ODE_trap_Lorenz.py`.

calculations. The ODE has the property that $x(t + \Delta t)$ is a function of $x(t)$. We call this function the *flow map* and write it as $\Phi(x, \Delta t)$:

$$x(t + \Delta t) = \Phi(x(t), \Delta t) \ .$$

The Runge Kutta time step computes x_{n+1} as a function of x_n. We call this the *time step* map and write it as $\Psi(x, \Delta t)$:

$$x_{n+1} = \Psi(x_n, \Delta t) \ . \tag{9}$$

The methods we have seen are (see (1) and (8)

$$\Psi(x, \Delta t) = \begin{cases} x + \Delta t f(x) & \text{(forward Euler)} \\ x_n + \Delta t f(x_n + \frac{1}{2}\Delta t f(x_n)) & \text{(trapezoid rule)} \ . \end{cases} \tag{10}$$

The two stage method has a more complicated Ψ, but is gets the answer with less work.

The accuracy of a time stepping method is determined by its *residual*, also called *local truncation error*. The *residual*, for Runge Kutta methods, is the difference between the time step map and the exact flow map. For technical reasons, the definition is scaled with a factor of Δt:

$$\Delta t \, R(x, \Delta t) = \Psi(x, \Delta t) - \Phi(x, \Delta t) \ . \tag{11}$$

The convergence theorem (proved below) concerns the maximum error up to some time T:

$$M(T, \Delta t) = \max_{t_n \leq T} \left| x^{(n)} - x(t_n) \right| \ . \tag{12}$$

Roughly, it says that if R is order Δt^p, then $M(T, \Delta t)$ also is order Δt^p. We are interested in $\Delta t \to 0$, so larger p means smaller error, or greater accuracy. The best (largest) p for a given method is its *order of accuracy*. The forward Euler method above has $p = 1$, which makes it *first order* accurate. The fancier trapezoid rule (5), (6), (7) has $p = 2$, which makes it *second order*. Figures 1 and 2 show the advantage of the fancier method.

2 ODE, the initial value problem

This section consists of a few points about differential equations. Much of it will be review to many people. Here, x_j refers to component j of $x \in \mathbb{R}^d$. Elsewhere, x_j may refer to the approximation to $x \in \mathbb{R}^d$ at time t_j.

2.1 The first order system

We often use a dot to denote derivatives with respect to time and a prime or a gradient symbol for derivatives in space, so

$$\frac{dx_j(t)}{dt} = \dot{x}_j \ , \quad \frac{df(x)}{dx} = f'(x) \ .$$

A *system* of differential equations is a set of differential equations that governs the evolution of d variables $x_1(t), \ldots, x_d(t)$. The system is *first order* if it specifies only first derivatives of the variables x_j. Such a system may be written in the form

$$\begin{aligned} \dot{x}_1(t) &= f_1(x_1(t), \ldots, x_d(t)) \\ &\vdots \qquad\quad \vdots \\ \dot{x}_d(t) &= f_d(x_d(t), \ldots, x_d(t)) \ . \end{aligned}$$

Usually, a system must be solved *simultaneously*. It is impossible to compute one of the components, say $x_1(t)$, without computing all of them. The system is generally written in vector form

$$\dot{x}(t) = f(x(t)) \ . \tag{13}$$

Here, $x(t)$ is the column vector with components $x_1, \ldots, x_d$ and $f(x)$ is the column vector whose components are f_j.

We might call (13) a *generic* first order system. Many dynamical systems have to be re-written to be put in this form (example below). Many dynamical systems have *special structure* that can be used to make the solution process faster or more accurate. Still, the generic system is useful because it exposes in a simple way things that are common to many dynamical systems, things that might be harder to spot in specific examples with special structure. Methods developed for generic problems are applicable to any problem that can be formulated in that way. Generic methods (generic mathematical analysis, generic software) may not be optimal for a problem with special structure, but they are available. The user can use a generic ODE software package and communicate her/his specific problem by writing a module that evaluates f.

Here is one example of formulating a dynamical system in the generic format (13). Newton's law $F = ma$ (force = mass times acceleration) leads to second order equation systems that may be written as

$$m\ddot{r} = F(r) \ . \tag{14}$$

In a simple case, $r = (r_x, r_y, r_z) \in \mathbb{R}^3$ could represent the cartesian coordinates of an object in three dimensional physical space. The force also has three components $F = (F_x(r), F_y(r), F_z(r))$. Physicists and engineers often use bold font, $\mathbf{r}$, or decoration, $\vec{r}$, to distinguish vectors from plain numbers or components. You can formulate the second order system (14) as a first order system by introducing the first time derivatives as new variables:

$$\begin{aligned} x_1(t) &= r_x(t) \\ x_2(t) &= r_y(t) \\ x_3(t) &= r_z(t) \\ x_4(t) &= \dot{r}_x(t) \\ x_5(t) &= \dot{r}_y(t) \\ x_6(t) &= \dot{r}_z(t) \ . \end{aligned}$$

Some of the first order differential equations come directly from this definition, as $\dot{x}_1 = \dot{r}_x = x_4$. The others come from the original dynamics, as $\dot{x}_4 = \ddot{r}_x = F_x(r)$. The differential equation system can be written as

$$\begin{aligned}
\dot{x}_1 &= f_1(x) = x_4 \\
\dot{x}_2 &= f_2(x) = x_5 \\
\dot{x}_3 &= f_3(x) = x_6 \\
\dot{x}_4 &= f_4(x) = F_x(x_1, x_2, x_3) \\
\dot{x}_5 &= f_5(x) = F_y(x_1, x_2, x_3) \\
\dot{x}_6 &= f_6(x) = F_z(x_1, x_2, x_3) \ .
\end{aligned}$$

A more typical example would have several particles with locations $r_1, \ldots, r_N$. Here, each r_j is a point in three dimensional space, so $R = (r_1, \ldots, r_N)$ has $3N$ coordinates in all. The force on particle j depends on the locations of all the other particles, so

$$m_j \ddot{r}_j = F_j(R) \ .$$

In generic form, this becomes a system of $6N$ first order equations for the positions and velocities of the particles.

The Python modules (files) in `ODE_solver` illustrate several points made here. This code simulates the motion of N particles moving in one dimension, which leads to a system of $2N$ first order differential equations. The specific system is the Fermi Pasta Ulam lattice.[4] There are three modules, a "main program", a "package" module, and a "user" module. The main program, `lattice.py` sets parameter values, manages the computation and makes the movie. The package module `EulerForward.py` implements one time step of the forward Euler method (1). This calls the user module `lattice_force.py`, which knows the specific problem. Figure 3 is a screen capture that shows a frame of the output movies from runs with a larger and smaller time step. The large time step calculation is much faster, and very inaccurate.

The *initial value problem* is to find $x(t)$ for $t \geq t_0$, subject to the *initial condition*

$$x(t_0) = x_0 \ . \tag{15}$$

The existence and uniqueness theorem for ODE guarantees that if f is *locally Lipschitz*[5], then there is a $t_1 > 0$ so that there is a unique *trajectory* (solution) of (13) and (15) defined for t in the range $t_0 \leq t < t_0 + t_1$. The example $f(x) = x^2$,

[4] Fermi Pasta and Ulam used this dynamical system in one of the earliest "automatic electronic computer" experiments. The results contradicted their expectations and some fundamental beliefs regarding statistical physics. The belief was that a generic nonlinear hamiltonian system would show "molecular chaos" and evolve quickly toward thermodynic equilibrium (of the microcannonical ensemble). Instead, the system exhibits recurrence – returning on a relatively short time scale to configurations similar to the starting configuration. This behavior still contradicts dogma and still is unexplained. Attempts to explain it led to discovery of nonlinear lattice systems that are integrable – the Toda lattice. The the Fermi Pasta Ulam lattice does not seem to be integrable.

[5] This means that for every $x \in \mathbb{R}^n$ there is a $\delta_x > 0$ and an L_x so that if $|x - y| \leq \delta_x$ and $|x - z| \leq \delta_x$, then $|f(y) - f(z)| \leq L_x |y - z|$. If f is continuously differentiable (the derivative

with $x(0) = 1$ and $x(t) = (1-t)^{-1}$ shows that the trajectory may not be defined for all $t > t_0$.

If $f(x)$ is differentiable for every x and[6] f' is continuous, then f is locally Lipschitz. If $|f'(x)| \to \infty$ as $x \to \infty$, then f is not *globally Lipschitz*[7]. For example, the Lorenz system of Section 1 is not globally Lipschitz. A linear system of equations, which means that $f(x) = Ax$ for some matrix A, is globally Lipschitz.

If f is not globally Lipschitz, then the solution can *blow up* at some time $T > t_0$. This means that the trajectory is defined for all t between the initial time t_0 and T, but that $|x(t)| \to \infty$ as $t \to T$. Many interesting and useful ODE systems have solutions that blow up. If they do, that tells us something interesting about the physical system they model. The physical system itself may not literally blow up, but the model says that something dramatic happens as t approaches T. The model probably stops being valid there. Nevertheless, the physical system may do something dramatic. We will always assume that f is locally Lipschitz.

---

exists for every x and the derivative is a continuous function of x), then f is locally Lipschitz. The function $f(x) = x^2 \sin(x^{-4})$ is differentiable at every point (the derivative at zero is zero), but the derivative is not a continuous function of x and f is not locally Lipschitz (there is no $\delta_0 > 0$ and L_0). If your ODE is $\dot{x} = x^2 \sin(x^{-4})$, then much of the theory in this chapter does not apply.

[6] We often write $f'(x)$ for the Jacobian matrix of first partial derivatives of f.

[7] This means that there is a single L so that $|f(x) - f(y)| \leq L\,|x - y|$ for all x and y.

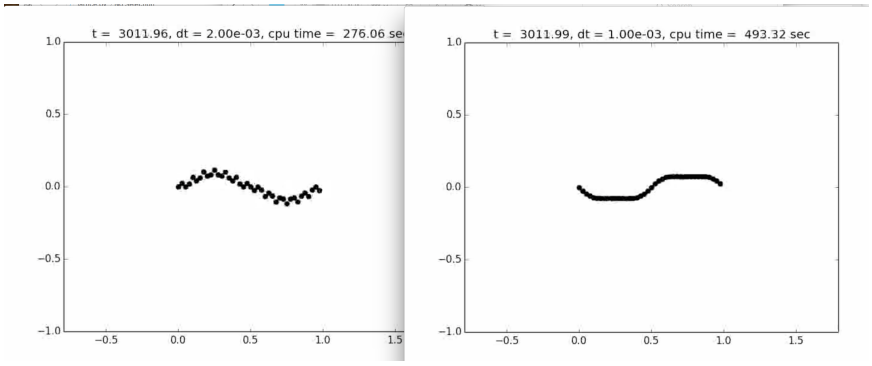

Figure 3: Simulation of a Fermi Pasta Ulam lattice using the code in `ODE_solver`. A screencap of the output movies for $\Delta t = .002$ (left) and $\Delta t = .001$ (right). They represent the computed solution at time approximately $t = 3012$. The $\Delta t = .002$ calculation took about 276 seconds on some laptop. The $\Delta t = .001$ calculation took about 493 seconds, which is something like twice as much time. It is clear from the pictures that the $\Delta t = .002$ calculation is fairly wrong.

2.2 The flow map and short time behavior

The solution $x(t_0 + t)$ is a function of x_0 and t. We call it the *flow map* if we're thinking of it as a function of x_0 with t fixed. We call it the *trajectory* if we're thinking of it as a function of t with x_0 fixed. The existence and uniqueness theorem for differential equations may be viewed as saying that for any $\overline{x}$ there is a $t_1 > 0$ and an $r > 0$ so that the flow map is defined for any $t < t_1$ and $|x_0 - \overline{x}| < r$. We can write the same thing as

$$x(t_0 + t) = \Phi(x(t_0), t) \; . \tag{16}$$

The formula $y = \Phi(x, t)$ means that if the trajectory is at x at some time, then after a time t it is at y. For example, if $d = 1$ and $f(x) = 2x$, then $\Phi(x, t) = e^{2t}x$. The differential equation $\dot{x} = f(x)$ implies that the flow map satisfies the differential equation

$$\frac{\partial}{\partial t} = \partial_t \Phi(x, t) = f(\Phi(x, t)) \; . \tag{17}$$

The flow map is defined for each x at least for sufficiently small t. But, for nonlinear differential equations it need not be defined for all t for a specific x. For example, consider the equation

$$\dot{x} = x^2 \; .$$

This differential equation has $f(x) = x^2$, which is locally Lipschitz but not globally Lipschitz (no L works for all y and z.) The solution to the initial value problem with $x(0) = x_0$ is

$$x(t) = \frac{1}{x_0^{-1} - t} \; .$$

This *blows up* (goes to infinity) as $t \to t_* = x_0^{-1}$, if $x_0 > 0$.

Runge Kutta methods are constructed using Taylor series expansion of $\Phi(x, t)$ as a function of t near $t = 0$. This is the Taylor series expansion of $x(t_0 + t)$, as a function of t, about $t = 0$. For convenience, we take $t_0 = 0$ in these derivations. The Taylor series is

$$x(t) = x(0) + t\dot{x}(0) + \frac{1}{2}t^2\ddot{x}(0) + \frac{1}{6}t^3\frac{d^3}{dt^3}x(0) + \cdots \; .$$

It is easier to write expressions like these if we leave out the argument 0 on the right side. For example,

$$x(t) = x + t\dot{x} + \frac{1}{2}t^2\ddot{x} + \frac{1}{6}t^3\frac{d^3}{dt^3}x + \cdots \; .$$

Here, and often in this course, we make the convention that if the argument is left out is it assumed to be zero.

The Taylor coefficients $\dot{x}$ (meaning $\dot{x}(0)$), $\ddot{x}$, etc., are found from the differential equation. The first such formula is just

$$\dot{x} = f(x) \; . \tag{18}$$

We find the second formula by differentiating the ODE formula with respect to t. In these calculations, we write in t when it is not zero.

$$\begin{aligned}
\ddot{x}(t) &= \frac{d}{dt}\dot{x}(t) \\
&= \frac{d}{dt} f(x(t)) \\
&= f'(x(t))\,\dot{x}(t) \qquad (19) \\
\ddot{x}(t) &= f'(x(t))\,f(x(t))\;. \qquad (20)
\end{aligned}$$

The main point of that calculation is the chain rule. It is convenient to write it in terms of components in the form

$$\frac{d}{dt} f_j(x(t)) = \sum_{k=1}^{d} \frac{\partial f_j(x(t))}{\partial x_k}\,\frac{dx_k(t)}{dt}\;. \qquad (21)$$

You can recognize this as the right side of (19), since f' is the Jacobian matrix of partial derivatives. Setting $t = 0$ in (20), we get

$$\ddot{x} = f'(x)f(x)\;. \qquad (22)$$

This formula suffices to derive the two stage second order method of Section 1.

Methods of order higher than 2 require derivatives beyond $\ddot{x}$. These are more complicated and are not easily expressed in simple matrix/vector notation. Therefore the calculations are presented with components and sums. The third derivative is found by differentiating the second derivative formula. We use the chain rule (as before), and the product rule:

$$\begin{aligned}
\frac{d^3}{dt^3}x_j(t) &= \frac{d}{dt}\ddot{x}_j(t) \\
&= \frac{d}{dt}\sum_{k=1}^{d}\left(\frac{\partial f_j(x(t))}{\partial x_k} f_k(x(t))\right) \\
&= \sum_{k=1}^{d}\left(\sum_{l=1}^{d}\frac{\partial^2 f_j(x(t))}{\partial x_k \partial x_l}\frac{dx_l(t)}{dt} f_k(x(t)) + \frac{\partial f_j(x(t))}{\partial x_k}\frac{\partial f_k(x(t))}{\partial x_l}\frac{dx_l(t)}{dt}\right) \\
&= \sum_{k=1}^{d}\left(\sum_{l=1}^{d}\frac{\partial^2 f_j(x(t))}{\partial x_k \partial x_l} f_k(x(t)) f_l(x(t)) + \frac{\partial f_j(x(t))}{\partial x_k}\frac{\partial f_k(x(t))}{\partial x_l} f_l(x(t))\right)
\end{aligned}$$

Some simplifications of notation make this formula easier to understand. The first simplification is to set $t = 0$ and leave out the t argument:

$$\frac{d^3}{dt^3}x_j = \sum_{k=1}^{d}\left(\sum_{l=1}^{d}\frac{\partial^2 f_j(x)}{\partial x_k \partial x_l} f_k(x) f_l(x) + \frac{\partial f_j(x)}{\partial x_k}\frac{\partial f_k(x)}{\partial x_l} f_l(x)\right)\;.$$

The second simplification is to write partial derivatives as subscripts, after a comma. For example,

$$\frac{\partial f_j(x)}{\partial x_k} = f_{j,k}\;,\quad \frac{\partial^2 f_j(x)}{\partial x_k \partial x_l} = f_{j,kl}\;,\quad \text{etc.}$$

With this, we have

$$\frac{d^3}{dt^3}x_j = \sum_{k=1}^{d}\left(\sum_{l=1}^{d} f_{j,kl}(x)f_k(x)f_l(x) + f_{j,k}(x)\partial f_{k,l}(x)f_l(x)\right) .$$

The third simplification is the *Einstein summation convention*, we leave out the sum symbols. The convention is that whenever you see a repeated index, you sum over that index running from 1 to d. You do this for an index that really is an index (such as k in f_k) or an index that represents a partial derivative (such as k in $f_{j,kl}$). There can be more than one summation in one formula Some examples are

$$\sum_{k=1}^{d} f_{j,k}f_k = f_{j,k}f_k$$

$$\sum_{k=1}^{d}\sum_{l=1}^{d} f_{j,kl}f_kf_l = f_{j,kl}f_kf_l$$

These simplifications give the more compact formula

$$\frac{d^3}{dt^3}x_j = f_{j,kl}(x)f_k(x)f_l(x) + f_{j,k}(x)f_{k,l}(x)f_l(x) . \tag{23}$$

This suffices to make a third order accurate three stage method.

A fourth order method requires the fourth derivative. The third derivative calculation may be written simply as:

$$f_{j,k}f_k \xrightarrow{\frac{d}{dt}} f_{j,kl}f_kf_l + f_{j,k}f_{k,l}f_l .$$

You differentiate using the product rule and chain rule, then you replace $\dot{x}_l$ with f_l. In this spirit, we calculate some of the terms in the fourth derivative:

$$\begin{aligned} f_{j,kl}f_kf_l \xrightarrow{\frac{d}{dt}} & f_{j,klm}f_kf_lf_m + f_{j,kl}f_{k,m}f_lf_m + f_{j,kl}f_kf_{l,m}f_m \\ = \; & f_{j,klm}f_kf_lf_m + 2f_{j,kl}f_{k,m}f_lf_m . \end{aligned}$$

You can see that the second and third terms are equal using the substitution $k \leftrightarrow l$. But you also see that these calculations are complicated, with many terms involving various combinations of derivatives. There is a reason most people don't read the derivations of Runge Kutta methods.

Altogether, these calculations may be written in the form

$$\begin{aligned} x_j(t) = \; & x_j \\ & + tf_j(x) \\ & + \frac{1}{2}t^2 f_{j,k}(x)f_k(x) \\ & + \frac{1}{6}t^3\left(\, f_{j,kl}(x)f_k(x)f_l(x) + f_{j,k}(x)f_{k,l}(x)f_l(x)\,\right) \\ & + O(t^4) . \end{aligned}$$

We will derive Runge Kutta methods using the same calculation written in terms of the flow map:

$$\begin{aligned}
\Phi_j(x,t) &= x_j && (24)\\
&+ t f_j(x) && (25)\\
&+ \frac{1}{2} t^2 f_{j,k}(x) f_k(x) && (26)\\
&+ \frac{1}{6} t^3 \left(f_{j,kl}(x) f_k(x) f_l(x) + f_{j,k}(x) f_{k,l}(x) f_l(x) \right) && (27)\\
&+ O(t^4) \; . && (28)
\end{aligned}$$

The matrix vector notation for $f_{j,k} f_k$ is $f' f$. One matrix vector notation for $f_{j,kl}(x) f_k(x) f_l(x)$ is $f''[f,f]$. In this notation, the expansion of Φ may be written

$$\begin{aligned}
\Phi(x,t) = x &+ t f(x) + \tfrac{1}{2} t^2 f'(x) f(x)\\
&+ \tfrac{1}{6} t^3 \left[f''(x)[f(x), f(x)] + \left(f'(x)\right)^2 f(x) \right] + O(t^4) \; .
\end{aligned}$$

2.3 Linear systems

This subsection reviews some of the properties of linear systems of differential equations. A linear system of ODE has the form

$$\dot{x} = Ax \; . \tag{29}$$

Here, A is a $d \times d$ matrix. Much of the behavior of solutions may be understood in terms of eigenvalues and eigenvectors of A. A number λ_j is an eigenvalue with right eigenvector r_j if

$$Ar_j = \lambda_j r_j. \tag{30}$$

The eigenvalue may be complex even when A is real. The eigenvector is complex if the eigenvalue is complex. The row vector l_j is the corresponding left eigenvector if

$$l_j A = \lambda_j l_j \; . \tag{31}$$

If r_k and l_k correspond to $\lambda_k \neq \lambda_j$, then

$$l_k r_j = 0 \; .$$

A matrix is *diagonalizable* if there is a basis of d linearly independent right eigenvectors. These may be arranged a matrix of right eigenvectors

$$R = \begin{pmatrix} | & | & \cdots & | \\ r_1 & r_2 & \cdots & r_d \\ | & | & \cdots & | \end{pmatrix} \; .$$

In this case, there exist linearly independent left eigenvectors that may be arranged into a left eigenvector matrix

$$L = \begin{pmatrix} - & l_1 & - \\ - & l_2 & - \\ & \vdots & \\ - & l_d & - \end{pmatrix} . \tag{32}$$

These may be chosen so that

$$LR = I . \tag{33}$$

This matrix equation is equivalent to the *bi-orthogonality* conditions

$$l_k r_j = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k. \end{cases}$$

If A is symmetric, then the eigenvalues are real, A is diagonalizable, and we may take

$$L = R^t .$$

In is the same as saying that the transpose of a right eigenvector is a left eigenvector (which is obvious if A is symmetric) and that the right eigenvectors may be chosen to be ortho-normal

$$r_k^t r_j = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k. \end{cases}$$

If A is diagonalizable, we create a diagonal eigenvalue matrix

$$\Lambda = \text{diag}(\lambda_1, \cdots, \lambda_d) .$$

The eigenvalue, eigenvector relations (30) may be written in the matrix form

$$AR = R\Lambda . \tag{34}$$

If we have R and Λ that satisfy the matrix eigenvalue eigenvector relation (34), we may define a new matrix $L = R^{-1}$ and multiply (34) on both sides by L. The result is

$$LA = \Lambda L .$$

This is the matrix version of the left eigenvalue eigenvector relations (31). The relations also may be written in the form

$$A = R\Lambda L \ (\text{ or } \ \Lambda = LAR \ ) \ , \quad LR = I . \tag{35}$$

The eigenvector basis transforms the coupled system (29) into a collection of uncoupled scalar differential equations. You can see this by multiplying both sides of (29) by L, using the relation $RL = I$, and defining

$$u = Lx . \tag{36}$$

The calculation is:

$$\begin{aligned} \dot{x} &= Ax \\ L\dot{x} &= LA\,(RL)\,x \\ \frac{d}{dt}\,(Lx) &= (LAR)\,(Lx) \\ \frac{d}{dt}u &= \Lambda u\ . \end{aligned}$$

This is equivalent to the d independent differential equations

$$\dot{u_j} = \lambda_j u_j\ .$$

The solution is

$$u_j(t) = e^{\lambda_j t} u_j(0)\ .$$

The eigenvector matrix R represents x in terms of u (since $R = L^{-1}$):

$$x = Ru\ .$$

This gives the eigenvalue eigenvector representation of the solution

$$x(t) = \sum_{j=1}^{d} e^{\lambda_j t} u_j(0)\, r_j\ . \tag{37}$$

This formula may be given in matrix form as

$$e^{tA} = Re^{t\Lambda}L\ . \tag{38}$$

Here, e^{tA} is the matrix exponential, or fundamental solution, that satisfies

$$x(t) = e^{tA}x(0)\ .$$

Also, $e^{t\Lambda}$ is the matrix exponential of the diagonal matrix, which is also diagonal with $e^{t\lambda_j}$ on the diagonal. You can check that

$$x(t) = Re^{t\Lambda}Lx(0)$$

is the same as both (37) and (38).

The definition of L (32) and the definition of u (36) imply that the components of u are

$$u_j = l_j x\ .$$

These u_j are the "expansion coefficients" when x is written in terms of the right eigenvector basis

$$x = \sum_{j=1}^{d} u_j r_j\ .$$

This is the components version of the matrix equation

$$x = RLx = Ru \; .$$

If A is symmetric, then $l_j = r_j^t$.

The *spectrum* of A is the set of eigenvalues of A. The spectrum tells you a lot about trajectories $x(t)$. Write an eigenvalue in terms of its real and imaginary parts as

$$\lambda_j = \mu_j + i\omega_j \; . \tag{39}$$

The real part of λ_j determines how trajectories grow:

$$\left| e^{\lambda_j t} \right| = e^{\mu_j t} \; .$$

These grow "exponentially" if $\mu_j > 0$, so λ_j is in the right half of the complex plane, and decay if $\mu_j < 0$ (left half plane). If $\mu_j = 0$, so λ_j is on the imaginary axis, then the solution neither grows nor decays. It oscillates if $\omega_j \neq 0$ and does nothing if $\omega_j = 0$ (so $\lambda_j = 0$). The *spectral abscissa* is

$$\mu(A) = \max_j \mu_j \; .$$

A generic trajectory grows or decays at the rate $\mu(A)$, which means

$$|x(t)| = Ce^{t\mu(A)} + \text{ smaller as } t \to \infty \; .$$

This can fail to happen, for example, if the expansion coefficients of initial conditions vanish. This is unlikely in exact arithmetic and even more unlikely in computer arithmetic.

This simple story does not cover all the cases you will encounter in numerical solution of differential equations. For one thing, there are matrices that cannot be diagonalized. We say these have *nontrivial Jordan structure*. For example, consider a one dimensional harmonic oscillator with real frequency ω:

$$\ddot{r} = -\omega^2 r \; .$$

This is written as a first order system using $x_1 = r$, $x_2 = \dot{r}$. The dynamical equations are $\dot{x}_1 = \dot{r} = x_2$, and $\dot{x}_2 = \ddot{r} = -\omega^2 r = -\omega^2 x_1$. Therefore

$$\frac{d}{dt} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \; .$$

The matrix

$$A = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}$$

has eigenvalues $\lambda_1 = i\omega$, and $\lambda_2 = -i\omega$. If $\omega \neq 0$ these are distinct so A may be diagonalized. The eigenvalues are on the imaginary axis and the solution formula (37) implies that $x(t)$ remains bounded as $t \to \infty$.

If $\omega = 0$ then A is a Jordan block of size 2 with eigenvalue 0. There is no representation of the solution in the form (37). In fact, a typical solution grows

linearly as $t \to \infty$. This is obvious from the physical problem – if $\omega = 0$, we have an oscillator with no spring (no restoring force). The dynamics are $\ddot{r} = 0$, which gives $r(t) = r(0) + t\dot{r}(0)$. The particle moves with a constant velocity.

Stability is a constant issue in numerical solution of differential equations. A simple stability question would be whether $\left\| e^{tA} \right\|$ is bounded as $t \to \infty$. A matrix like this would be called *stable*. This is true if $\mu(A) < 0$ and false if $\mu(A) > 0$ (the spectral abscissa). But if $\mu(A) = 0$, it depends on whether A has nontrivial Jordan structure corresponding to pure imaginary eigenvalues. If the eigenvalues on the imaginary axis are simple, as they are when $\omega \neq 0$ above, then A is stable. The theorem is that A is stable if all its eigenvalues are in the left half plane or the imaginary axis, and if the eigenvalues on the imaginary axis are simple.

The stability issue arises also for discrete time dynamics

$$x_{n+1} = Ax_n \ , \quad x_n = A^n x_0 \ .$$

The matrix A is stable for discrete time dynamics if all the eigenvalues of A are inside the unit disk and the ones on the unit circle (the boundary of the unit disk, the borderline eigenvalues) are simple. If A has a rank 2 Jordan structure for an eigenvalue on the unit circle, then (generically) the iterates grow linearly in time. For rank p Jordan blocks, the growth is on the order of n^{p-1}.

2.4 Perturbation, linearization, sensitivity

Sensitivity is the question: how does the output change if you change the input a little. It is important physically for many reasons. It is important in numerical solution of differential equations because it determines the condition number of the ODE problem. A principle that applies to solving differential equations is that a problem that ill conditioning limits the accuracy you should expect to get.

For the initial value problem, we ask how $x(t)$ changes if we change $x(t_0)$ a little. This could be written that some people (including myself) find confusing:

$$\frac{\partial x_i(t)}{\partial x_j(t_0)} \ .$$

Instead, we write this as the derivative (Jacobian) matrix of the flow map

$$\Phi'_{ij}(x,t) = \frac{\partial \Phi_i(x,t)}{\partial x_j} = \partial_{x_j} \Phi_i(x,t) \ .$$

We wrote Φ'_{ij} as $\Phi_{i,j}$ just above, but matrix indices are usually written without the comma. Mathematical notation never stays consistent. The $d \times d$ Jacobian matrix $\Phi'(x,t)$ describes the sensitivity of the solution at time t to perturbations in the initial condition.

The sensitivity matrix Φ' may be computed by solving a system of differential equations. These equations are derived by using the fact that partial derivatives

commute. We differentiate the differential equation (17) with respect to x_j, and the result is (using the chain rule to differentiate $f_j(\Phi(x,t))$ as above)

$$\begin{aligned}\partial_t \partial_{x_j} \Phi_i(x,t) &= \partial_{x_j} \partial_t \Phi_i(x,t) \\ &= \partial_{x_j} f_i(\Phi(x,t)) \\ &= \partial_{x_k} f_i(\Phi(x,t)) \partial_{x_j} \Phi_k(x,t) \ .\end{aligned}$$

In index notation with the Einstein convention, this is written

$$\partial_t \Phi_{i,j}(x,t) = f_{i,k}(\Phi(x,t)) \Phi_{k,i}(x,t) \ .$$

We concentrate on the sensitivity of a specific trajectory $x(t)$. We drop the x dependence and write

$$M(t) = \Phi'(x(0),t) \ .$$

The dynamics of M are

$$\partial_t M(t) = f'(x(t))\, M(t) \ . \tag{40}$$

This is the x dynamics (13) linearized about the trajectory $x(t)$. You can compute the trajectory and its sensitivity by solving the ODE system that combines (13) and (40). The initial condition for M is $M(0) = I$, which is the Jacobian of $\Phi(x,0) = x$.

Many systems show extreme sensitivity to perturbations in the initial conditions. This is possible even when the trajectories themselves are bounded. Even then it is possible that typical bounded trajectories have

$$\|M(t)\| \sim e^{\mu t}$$

as $t \to \infty$, with $\mu > 0$. This μ is the *principal Lyapunov exponent*. When μ is positive, then the system has "exponential" sensitivity to changes in the initial condition. Systems like that (bounded trajectories, positive Lyapunov exponent) are called *chaotic*. The Lorenz system (4) is an example.

A chaotic system is ill conditioned and therefore hard or impossible to solve accurately. The condition number is a dimensionless measure of sensitivity

$$\kappa = \left\| \frac{\partial \, \text{answer}}{\partial \, \text{data}} \right\| \cdot \frac{\|\text{data}\|}{\|\text{answer}\|}$$

Here, the data is x, the answer is $x(t) = \Phi(x,t)$, and the sensitivity is $M(t)$. Therefore,

$$\kappa(t) = \|M(t)\| \, \frac{\|x\|}{\|\Phi(x,t)\|} \ .$$

A chaotic system has $\|M(t)\| \sim e^{\mu t}$ while the fraction $\|x\| \, / \, \|\Phi(x,t)\|$ does not go to zero as $t \to \infty$. Therefore

$$\kappa(t) \sim e^{\mu t} \quad , \quad \text{as } t \to \infty.$$

The *Liapunov time* (a term used by physicists) is

$$T_L = \frac{1}{\mu} .$$

It represents the time needed for perturbations to grow roughly by a factor of e. After about 60 Lyapunov times, perturbations grow by a factor of about $e^{60} \approx 10^{26}$. This means that a perturbation in the initial data of size 10^{-26} is amplified to become a perturbation of order 1 at time $t = 60T_L$. From a practical point of view, this means that the solution at time $60T_L$ cannot be computed "accurately" in the sense that it is not possible to predict the numbers $x(t)$ accurately. For one thing, initial condition $x(0)$ is perturbed by a larger 10^{-17} when it is rounded to double precision floating point. But even with extended precision arithmetic, it is unlikely that an initial value would be known with 26 digits of accuracy.

Numerical computing can be useful even beyond the $60T_L$ limit. Although $x(t)$ cannot be calculated, there may be a *quantity of interest*, or *QOI*, that can

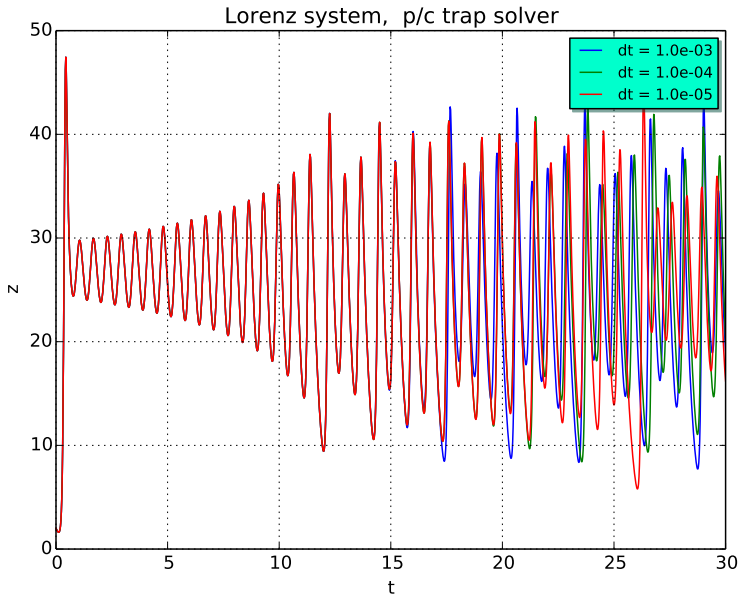

Figure 4: The Lorenz system (4) solved using the same trapezoid rule but for a longer time and with much smaller time steps. Chaos starts to set in around $T = 17$. By time $T = 30$, even the smallest Δt solution is inaccurate. The Python source is in `codes/LorenzChaos.py`.

be. An example may be the average value of some "observable" $q(x)$:

$$Q = \frac{1}{T} \int_0^T q(x(t))\, dt \ .$$

In climate science, computing $x(t)$ would be "predicting the weather". But averages like Q are "climate" – average rainfall per year, etc. It may be possible to predict climate without being able to predict the weather. An example (not weather and climate, unfortunately) is in the exercises.

The weather and climate problem exists in almost any scientific computation. The theory may predict that your QOI is given correctly in the limit $\Delta t \to 0$, but that limit may be inaccessible with your computing hardware and algorithms. Practical people often attempt a *convergence study*, which means comparing the results from several runs with different Δt (or other computational parameters in more complex computations). It can be distressing that $x(t)$ is not converging (yet). Nevertheless, some QOI may seem to be "converged" (computed as accurately as needed). This is a very important aspect of scientific computing, and one that is often under-estimated or even neglected by people in a hurry. Such people often have the wrong answer without knowing it.

3 Time stepping and Forward Euler

Time stepping means computing a trajectory (an approximation of a trajectory) through a sequence of small *time steps*. A time step *advances* the solution from time t_n to time $t_{n+1} = t_n + \Delta t$. The approximate solution is

$$x_n \simeq x(t_n) \ .$$

Runge Kutta methods are methods in which x_{n+1} depends on x_n alone. *Multistep* methods compute x_{n+1} using x_n and earlier values x_{n-1}, etc. The time step is some formula or algorithm that takes x_n and Δt and produces x_{n+1}. This will be denoted by Ψ, so

$$x_{n+1} = \Psi(x_n, \Delta t) \ .$$

Different methods have different functions Ψ. The time step Ψ is supposed to mimic the actual flow map Φ defined in Subsection 2.2.

The *forward Euler* method is

$$x_{n+1} = x_n + \Delta t f(x_n) \ . \tag{41}$$

This has the time step function

$$\Psi(x, \Delta t) = x + \Delta t f(x) \ .$$

Figure 1 has a plot of approximate solutions to a simple ODE system computed by forward Euler for a decreasing sequence of Δt values. The approximations

(seem to) converge as $\Delta t \to 0$. There is a lot of theory that explains and predicts the size of the error and how it grows with t. It will enable us to find methods that are far better than forward Euler.

Here is a version of this convergence theory, as applied to the forward Euler method. The theory is a relation between *residual* (or *local residual*) and *error* (or *global error*). Throughout numerical computing, *resudual* refers to the amount by which some equations defining the answer are not satisfied, while *error* refers to the difference between the computed answer and the actual answer. A method is *stable* if there is a bound for error in terms of residual. The point of this approach is that residual is not so hard to estimate or measure. Measuring the error directly would require knowing the answer. But the point of computing is to find unknown answers.

The theory starts with terminology and notation. The *error* at time t_k is

$$E_n = x_n - x(t_n) \ .$$

The maximum error up to time t is

$$M(t, \Delta t) = \max_{t_k \leq t} |x_k - x(t_k)| \ . \tag{42}$$

A method is *convergent* if $M(t, \Delta t) \to 0$ as $\Delta t \to 0$. The definition (42) refers to a physical time t rather than a number of time steps n. As $\Delta t \to 0$, the number of time steps needed to reach a given physical time goes to infinity. It is unlikely that $M(t, \Delta t) \to 0$ uniformly as $t \to \infty$. If the problem is chaotic, we expect something like $M \sim \Delta t^p e^{\mu t}$. If you set $M = 1$ and solve for t (to find the time at which the method has lost all accuracy), you find

$$t = \frac{p}{\mu} \log(\Delta t^{-1}) \ .$$

A smaller Δt gets you further, but not by much.

A method has *order of accuracy* p, or is p^{th} order accurate, if

$$M(t, \Delta t) \leq C(t) \Delta t^p \ .$$

Normally, saying a method has order of accuracy p implies that it does not have order q for any $q > p$. The forward Euler method (41) is is convergent and first order accurate. Fancier methods described below have $p > 1$, they are *higher order*.

A technical point: we use absolute value signs in (42) and elsewhere to

represent any norm in the space $\mathbb{R}^n$. For example,

$$|x| = \sum_{j=1}^{d} |x_j|$$

or

$$|x| = \left(\sum_{j=1}^{d} x_j^2 \right)^{1/2}$$

or

$$|x| = \max |x_j|$$

etc.

The constants in the theorems may depend on which norm we use, but the theorems and the existence of constants does not. It is a theorem of mathematical analysis that in d dimensions, any two norms are related by a constant (technically, are *equivalent*). The theorems are simpler to state and easier to prove and more general if you allow $|\cdot|$ to represent any norm.

The *local truncation error*, or *residual*, or *one step error*, of a one-step method is error produced in a single time step. It simplifies the calculations later to include a factor of Δt in the definition, so

$$R(x, \Delta t) = \Delta t^{-1} \left[\Psi(x, \Delta t) - \Phi(x, \Delta t) \right] .$$

This definition allows the difference between Ψ and Φ to be written in terms of the local truncation error as

$$\Psi(x, \Delta t) = \Phi(x, \Delta t) + \Delta t R(x, \Delta t) . \tag{43}$$

It allows us to express the action of the time step method in terms of the exact solution to the differential equation

$$x_{n+1} = \Psi(x_n, \Delta t) = \Phi(x_n, \Delta t) + \Delta t R(x_n, \Delta t) .$$

The local truncation error, R, can be thought of as an extra "force" that the time stepping method adds to the differential equation. In this view, the difference between $x(t_n)$ and x_n is something like the difference between the exact solution x and the solution to the modified differential equation

$$\dot{y} = f(y) + R(t) . \tag{44}$$

Indeed, the forward Euler approximation to the modified equation (44) is

$$y_{n+1} = y_n + \Delta t \left[f(y_n) + R(t_n) \right] .$$

The definition (43), with the Δt pulled out, makes this interpretation possible.

Here is the main convergence theorem for Runge Kutta methods. Several assumptions are necessary, and these should seem natural by now.

1. To prove that the method converges up to time t, the true solution must be defined (not blown up) up to that time. We assume that $x(t)$ exists and satisfies the differential equation for all t in the range $0 \leq t \leq T$.

2. The residual defined by (43) must be small for any y that is close to the exact trajectory for $t \leq T$. We assume that there is an $r > 0$ and an a so that if $|y - x(t)| \leq r$ for some $t \leq T$ then

$$|R(y, \Delta t)| \leq a \Delta t^p \ .$$

3. The method is built from Δt and f in some natural way, specifically that $\Psi(x, \Delta t) = x + \Delta t \cdot (\text{something involving } f)$. We assume that that there is a b so that if $|y - x(t)| \leq r$ and $|z - x(t)| \leq r$ for some $t \leq T$, then

$$|\Psi(y, \Delta t) - \Psi(z, \Delta t)| \leq |y - z| + b \Delta t \, |y - z| \ .$$

   The forward Euler method (and all the other methods considered here) satisfy this if f is Lipschitz continuous.

Theorem. If hypotheses 1-3 are satisfied and $t \leq T$, then there is an $\epsilon > 0$ so that if $\Delta t \leq \epsilon$, then

$$M(t, \Delta t) \leq \frac{a \Delta t^p}{b} \left(e^{tb} - 1 \right) \ . \tag{45}$$

Remarks:

1. The error is allowed to grow exponentially in t, as we expect and as happened in the Lorenz system computation.

2. The right side is the solution to the model error propagation problem

$$\dot{m} = bm + a \Delta t^p \quad , \quad m(0) = 0 \ .$$

   The parameter b is related to the Lipschitz constant of f, which is the exponential growth rate of exact solutions.

3. The role of ϵ is that if $\Delta t \leq \epsilon$ and the conclusion (45) holds, then $|x_n - x(t_n)| \leq r$ so the bounds apply.

Proof. The proof is by induction on the time step n. The induction hypothesis is that

$$|x_n - x(t_n)| \leq \frac{a \Delta t^p}{b} \left(e^{bt_n} - 1 \right) \ .$$

We must prove this for $n + 1$. The main calculation is

$$\begin{aligned} x_{n+1} - x(t_{n+1}) &= \Psi(x_n, \Delta t) - \Phi(x(t_n), \Delta t) \\ &= [\Psi(x_n, \Delta t) - \Psi(x(t_n), \Delta t)] + [\Psi(x(t_n), \Delta t) - \Phi(x(t_n), \Delta t)] \ . \end{aligned}$$

The first term is controlled by the stability hypothesis 3. Specifically

$$\begin{aligned} |\Psi(x_n, \Delta t) - \Psi(x(t_n), \Delta t)| &\leq |x_n - x(t_n)| + b \Delta t \, |x_n - x(t_n)| \\ &\leq (1 + b \Delta t) \, M(t_n, \Delta t) \ . \end{aligned}$$

The second term is controlled by the consistency hypothesis 2. Specifically, in view of (43)

$$|\Psi(x(t_n), \Delta t) - \Phi(x(t_n), \Delta t)| \leq \Delta t \, a \Delta t^p \ .$$

The error amplification bound follows from combining these estimates:

$$M(t_{n+1}, \Delta t) \leq (1 + b\Delta t) \, M(t_n, \Delta t) + \Delta t \, a \Delta t^p \ . \tag{46}$$

It is an exercise (literally, exercise 2) to prove the error bound (45) from this.

4 Higher order methods

To review, a one-step method takes a sequence of time steps $x_{n+1} = \Psi(x_n), \Delta t)$. The accuracy of the method is determined by the local truncation error (43). This section describes some *explicit*, *multi-stage* methods that have a higher order of accuracy. We do not yet give a systematic theory of higher order Runge Kutta methods.

We start with second order two-stage methods. Some of these may be understood as *trapezoid rule* or *midpoint rule* approximations to an integral, or as *centered difference* approximations to derivatives. For example, here is a second order centered approximate derivative:

$$\dot{x}(t + \tfrac{1}{2}\Delta t) = \frac{x(t + \Delta t) - x(t)}{\Delta t} + O(\Delta t^2) \ .$$

Solving for the value at $t + \Delta t$, we have

$$x(t + \Delta t) = x(t) + \Delta t f(x(t + \tfrac{1}{2}\Delta t)) + O(\Delta t^3) \ . \tag{47}$$

This formula is like the forward Euler approximation, except that $f(x(t))$ is replaced by the unknown $f(x(t + \frac{1}{2}\Delta t))$. This would have order of accuracy $p = 2$, if it could be implemented.

There is a two stage *predictor corrector* version of this that is explicit and preserves the second order accuracy. For first step is to predict $x(t + \frac{1}{2}\Delta t)$ using forward Euler. We call the predicted change y_1:

$$y_1 = \tfrac{1}{2}\Delta t f(x(t)) \ .$$

This value is then used on the right side of (47):

$$y_2 = \Delta t f(x(t) + y_1) \ .$$

The eventual time step is

$$x_{n+1} = x_n + y_2 \ .$$

It is traditional to work with k_1 instead of y_1, with $y_1 = \Delta t k_1$. The time step map $\Psi(x, \Delta t)$ for this method is

$$k_1 = f(x) \tag{48}$$

$$y_2 = \Delta t f(x + \tfrac{1}{2}\Delta t k_1) \tag{49}$$

$$\Psi(x) = x + \Delta t k_2 \ . \tag{50}$$

The first stage (48) uses f to evaluate k_1. The second stage (49) uses f to evaluate k_2. The final assembly (49) constructs the new x value as xR plus linear combination of k_1 and k_2. This is the method used in the Lorenz calculations of Figures 2 and 4.

The general two stage explicit Runge Kutta method is

$$k_1 = f(x) \tag{51}$$
$$k_2 = f(x + \Delta t\, a_{12} k_1) \tag{52}$$
$$\Psi(x) = x + \Delta t\, (b_1 k_1 + b_2 k_2) \ . \tag{53}$$

The general three stage method is

$$k_1 = f(x) \tag{54}$$
$$k_2 = f(x + \Delta t\, a_{12} k_1) \tag{55}$$
$$k_3 = f(x + \Delta t\, (a_{21} k_1 + a_{22} k_2)) \tag{56}$$
$$\Psi(x) = x + \Delta t\, (b_1 k_1 + b_2 k_2 + b_3 k_3) \ . \tag{57}$$

It is possible to define r stage explicit methods in this way. The coefficients a_{jk} and b_k define the specific method. There are often displayed on the page in a specific way called the *Butcher tableau*.

We find the order of accuracy of a Runge Kutta method by calculating the Taylor series expansion of Ψ and comparing it to the expansion of the flow map Φ. The calculation for the two stage method is not complicated. We expand the second stage formula (52) and substitute the first stage formula (51) to get

$$\begin{aligned} k_2 &= f(x) + \Delta t f'(x) a_{12} k_1 + O(\Delta t^2) \\ &= f(x) + \Delta t a_{12} f'(x) f(x) + O(\Delta t^2) \ . \end{aligned}$$

At this point we drop the x arguments in these calculations. The assembly formula (53), is

$$\begin{aligned} \Psi(x, \Delta t) &= x + \Delta t\, (b_1 k_1 + b_2 k_2) \\ &= x + \Delta t\, (b_1 f + b_2\, (f + \Delta t a_{12} f' f)) + O(\Delta t^3) \\ &= x + \Delta t\, (b_1 + b_2)\, f + \Delta t^2 b_2 a_{12} f' f + O(\Delta t^3) \ . \end{aligned}$$

We compare this with the short time expansion of the flow map Φ. From the Δt term (25) we see that the method is at least first order accurate if the $O(\Delta t)$ terms agree, which is

$$1 = b_1 + b_2 \ . \tag{58}$$

Comparing the Δt^2 terms in (26), we see that the method is at least second order accurate if

$$\frac{1}{2} = b_2 a_{12} \ . \tag{59}$$

It turns out (we will see) that it is not possible to match the Δt^3 terms and get a third order method. The method is second order accurate if the three

parameters b_1, b_2, and a_{12} are chosen to satisfy the two equations (58) and (59). This is two equations for three unknowns, so there should be a one parameter family of solutions. This is the family of second order two stage explicit Runge Kutta methods. One solution is $b_2 = 1$, $a_{12} = \frac{1}{2}$ and $b_1 = 0$. This is the predictor-corrector midpoint rule we used before. Another is $b_1 = b_2 = \frac{1}{2}$ and $a_{12} = 1$.

5 Error analysis and software validation

A code that has not been validated is wrong. Every software effort must include plans for validation. *Richardson estimation* is the basis of a common validation method, particularly for algorithms with high order accuracy. Richardson estimation is based on *asymptotic error expansions*, which are common in basic methods for differentiation and integration.

Suppose h is a step size parameter that is going to zero. Suppose the algorithm is estimating a quantity (the "answer") A_0 using the estimate $A(h)$. An asymptotic error expansion is a relation of the form

$$A(h) = A_0 + h^p A_1 + h^q A_2 + \cdots + O(h^r) \ . \tag{60}$$

Here, p is the order of accuracy of the method and $p < q < \cdots < r$. The one sided difference approximation

$$\dot{x}(t) \approx \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

has $p = 1$ (it's first order accurate) and $q = 2$ (a Taylor series calculation shows this). Here, we replaced the generic step size parameter with the specific parameter Δt for this problem. The centered difference approximation

$$\dot{x}(t) \approx \frac{x(t + \Delta t) - x(t - \Delta t)}{2\Delta t}$$

has $p = 2$ (it's second order accurate) and $q = 4$.

Suppose you want to validate a code that computes $A(h)$ and is supposed to have an asymptotic error expansion. One approach is to apply it to a problem where A_0 is known, a *test problem*. For this to be possible, the code must be written in a flexible modular way so that it is easy to apply it to different problems. You compute $A(h)$ and $A(2h)$ by doing two runs. You then solve for p as follows. The $2h$ result satisfies

$$A(2h) = A_0 + 2^p h^p A_1 + 2^q h^q A_2 + \cdots + O(h^r) \ .$$

Combine this with $A(h)$ and use (60), and you get

$$\frac{A(2h) - A_0}{A(h) - A_0} = 2^p + O(h^{q-p}) \ . \tag{61}$$

Simply put, if there is an asymptotic error expansion for a method of order p, then doubling the step size increases the error roughly by a factor of 2^p. To do a *convergence study*, you print the ratio (61) for a decreasing set of h values and check that the result converges to 2^p. That means 2 for a first order method, 4 for second order, 8 for third order, etc. This is particularly useful for checking high order methods that only are high order if you use the correct coefficients. The method may converge if you have a coefficient wrong, but it's unlikely to have the desired order of accuracy. If you don't know A_0, you can still compute a Richardson ratio like (61), but it would be

$$\frac{A(4h) - A(2h)}{A(2h) - A(h)} \; .$$

Many time stepping methods for ODE solving have asymptotic error expansions. See exercise 4 for an example. An important detail is to take the correct number of time steps. If you use n time steps of size Δt to get to time T, you must use exactly $n/2$ steps to get to time T with steps of size $2\Delta t$. This is possible if n is even. If you compute the number of time steps with the formula $n = T/\Delta t$, rounding may give you a number that is off by 1.

6 Exercises and examples

1. You can check the Taylor series calculations in Subsection 2.2 by checking that they give the right answer for one component systems

$$\dot{x} = f(x) \; , \quad \ddot{x} = f'(x)f(x) \; , \quad \frac{d^3}{dt^3}x = f''(x)f^2(x) + \left(f'(x)\right)^2 f(x) \; \ldots \; .$$

   Check that this is consistent with the terms given in Subsection 2.2. Calculate the next term here (hint: one of the terms is $f'''(x)f^3(x)$). If you have the energy, calculate the term by finishing the calculation started in Subsection 2.2 and show that these also are consistent.

2. The forward Euler method applied to $\dot{m} = bm + c$ is

$$m_{n+1} = (1 + \Delta t b)\, m_n + \Delta t \, c \; .$$

   Show that this always under-estimates the solution if $b > 0$ and $a > 0$ and $m(0) > 0$. That is, show that the hypothesis imply that

$$m(t) \geq (1 + tb)\, m(0) + tc \; .$$

   Use this to finish the induction step in the proof of (45).

3. Consider the linear ODE system (29) where A is a $d \times d$ matrix. The flow map for a linear ODE system is a linear map. The matrix that represents this map, $S(t)$, is called the *fundamental solution* or the *solution operator*:

$$\Phi(x_0, t) = S(t)x_0 \; .$$

The fundamental solution satisfies the matrix differential equation

$$\dot{S}(t) = AS(t) \; . \tag{62}$$

The initial condition is that at time zero we have the identity matrix:

$$S(0) = I \; . \tag{63}$$

(a) Show that the short time Taylor expansion of the fundamental solution is

$$S(t) = I + \Delta t A + \Delta t^2 \tfrac{1}{2} A^2 + \Delta t^3 \tfrac{1}{6} A^3 + \cdots \; .$$

Find the formula for the rest of the terms, which are familiar from the power series expansion of the exponential.

(b) Show that taking the terms up to and including Δt^p yields a time step method for this linear ODE system of order p. The method is $x_{n+1} = Q_p x_n$, which more explicitly is

$$x_{n+1} = \left(I + \Delta t A + \cdots + \Delta t^p \frac{1}{p!} A^p \right) x_n \; .$$

(c) Show that there is a Horner's rule factorization of Q_p of the form

$$Q_p = I + \Delta t A \left(I + \tfrac{1}{2} \Delta t A \left(I + \cdots \right) \cdots \right) \; .$$

(d) Give an algorithm for evaluating $Q_p x$ that involves p matrix vector multiplies involving A. Interpret this as a p stage Runge Kutta method of order p. The algorithm should have the form that is simplified from the general Runge Kutta multi-stage form (identify the coefficients $a_2, \cdots a_p, b$):

$$\begin{aligned} k_1 &= Ax \\ k_2 &= A\left(x + \Delta t \, a_2 k_1 \right) \\ k_3 &= A\left(x + \Delta t \, a_3 k_2 \right) \\ &\vdots \\ Q_p x &= x + \Delta t \, b \, k_p \; . \end{aligned}$$

(e) Show that this algorithm has order p of accuracy when applied to the linear ODE $\dot{x} = Ax$.

(f) This is sometimes called a *low storage* method. Show that this method can be implemented using storage for only two vectors of size d if matrix vector multiply can be done with storage for a single vector. *Remark:* Low storage can be significant when d is comparable to the machine memory size.

(g) Consider the nonlinear version for nonlinear ODE $\dot{x} = f(x)$:

$$\begin{aligned} k_1 &= f(x) \\ k_2 &= f(x + \Delta t\, a_2 k_1) \\ k_3 &= f(x + \Delta t\, a_3 k_2) \\ &\vdots \\ \Psi(x, \Delta t) &= x + \Delta t\, b\, k_p \ . \end{aligned}$$

Show that this method is second order accurate in general.

4. This exercise serves two purposes. One is to practice the convergence proof above. The other is to look at error expansions. Consider the forward Euler method (1) and suppose that the computed solution has the form

$$x_n = x(t_n) + \Delta t\, y(t_n) + \Delta t^2\, z_n \ .$$

The two goals are to find a differential equation for y that proves (from the existence theorem for differential equations) that the leading term in the error is proportional to Δt, and to show that z_n is bounded uniformly as $\Delta t \to 0$ with $t_n \le t$.

(a) Suppose $x(t)$ and $y(t)$ are smooth functions of t, so that $x(t_{n+1}) = x(t_n) + \Delta t \dot{x}(t_n) + \frac{1}{2}\Delta t^2 \ddot{x}(t_n) + O(\Delta t^3)$, and the same for y. Plug this into the difference equation (1) and equate the Δt terms on both sides. The result should be (something like)

$$\dot{y} = f'(x(t))y + (**)f'(x(t))f(x(t)) \ \ , \quad y(0) = 0 \ .$$

This doesn't yet prove anything, but it does suggest what the error looks like.

(b) Define $u_n = x(t_n) + \Delta t y(t_n)$, where $y(t)$ satisfies the ODE of part (a). Show that

$$u_{n+1} = u_n + \Delta t f(u_n) + \Delta t R_n \ ,$$

where

$$R_n = O(\Delta t^2) \ .$$

(c) Use part (b) to show that

$$|x_n - u_n| \le C(t_n)\Delta t^2 \ ,$$

for t less than the blow up time (as in the Theorem above, $C(t)$ is independent of Δt). Conclude that the numerical solution satisfies

$$x(t) = x(t_n) + \Delta t y(t_n) + O(\Delta t^2) \ .$$

5. Find a set of coefficients for a three stage explicit Runge Kutta method that make it third order.

6. Download the three Python modules used to make the Fermi Pasta Ulam movie. As you modify this code, try to keep the same coding style and conventions. Feel free to post your opinions of the coding and propose better coding conventions.

   (a) Modify the module that implements the forward Euler time step to do a time step of a three stage third order accurate Runge Kutta method. You may wish to look at the file `ODE_trap_Lorenz.py`, which implements a two stage second order method.

   (b) Apply your code to a simple problem with a known answer (nonlinear, $d > 1$, but simple and solvable) to computationally verify the third order accuracy.

   (c) Choose one of the two computations:

      i. Try to make the Fermi Pasta Ulam movie using this code with a larger time step. How much less CPU time does it take. You may wish to modify the code that decides when to make the next movie frame so that the frame times are not subject to accumulating rounding errors.

      ii. Solved the linearized perturbation equations for the Lorenz system and make a plot of

      $$\log\left(\|M(t)\|\right)$$

      as a function of t. It should eventually grow linearly, so $\|M(t)\|$ grown exponentially. Use this plot to estimate the principal Lyapunov exponent and to show that the Lorenz system is chaotic. You will have to run the code far beyond $T = 30$. You will either have to choose T so that $M(t)$ does not overflow, or do something to renormalize away the overflow.