15 January 2025 – PHYS 6260
Prof. John Wise
Newman Chapter 8: Ordinary Differential Equations

Perhaps the most common task for computational physics is the solution of differential equations. There are many methods that we can use to solve them, and we will cover them in this chapter.

8.1: 1st-order Differential Equations with One Variable

An ordinary differential equation has only one independent variable, such as time. It may contain other variables that are dependent on this independent variable, though. The simplest type of ordinary differential equation (ODE) is a first-order equation with one dependent variable, for example

$$\frac{dx}{dt} = \frac{2x}{t}, \tag{1}$$

which can be easily solved analytically by separating the variables. However, there can easily exist an ODE that isn't separable, such as

$$\frac{dx}{dt} = \frac{2x}{t} + \frac{3x^2}{t^3}, \tag{2}$$

which is also non-linear. We can solve this numerically. In general, a first-order ODE has the form

$$\frac{dx}{dt} = f(x,t), \tag{3}$$

where time t is the independent variable. Of course, there exists forms of ODEs with other independent variables, such as another spatial variable y or z. But for our purposes, we will focus on time-independent solutions. Furthermore to compute a solution (analytically or numerical), we need an initial condition at $t = 0$.

Euler's method

The most straightforward method is Euler's method that evolves x with its derivative evaluated at time t with some step-size h in time. First, we can write the Taylor expansion around time t to calculate the next value of x at time $t + h$,

$$x(t+h) = x(t) + h\frac{dx}{dt} + \tfrac{1}{2}h^2\frac{d^2x}{dt^2} + \dots \tag{4}$$

$$= x(t) + h\frac{dx}{dt} + O(h^2). \tag{5}$$

For Euler's method, we neglect all terms with order higher than h^2, giving

$$\boxed{x(t+h) = x(t) + h\frac{dx}{dt}} \tag{6}$$

Example 8.1: Euler's method

Let's use Euler's method to solve the ODE:

$$\frac{dx}{dt} = -x^3 + \sin t \tag{7}$$

with the initial condition of $x = 0$ at $t = 0$. Here is a program that performs the calculate from $t = 0 \to 10$ with 1000 steps:

```
from math import sin
import numpy as np
import matplotlib.pyplot as plt

def f(x,t):
    return -x**3 + sin(t)

a = 0.0           # Start of the interval
b = 10.0          # End of the interval
N = 1000          # Number of steps
h = (b-a)/N       # Size of a single step
x = 0.0           # Initial condition

tpoints = np.arange(a,b,h)
xpoints = np.zeros(N)
xpoints[0] = x
#
# To be completed in class: Loop over points to calculate x
#

plt.plot(tpoints,xpoints)
plt.xlabel("t")
plt.ylabel("x(t)")
plt.show()
```

Running this program gives a smooth plot that is a good approximation to the actual solution. In general, Euler's method is not a bad one, and in many cases, it is quite accurate. However, it is not widely used because it requires very little extra effort to devise a more accurate method, called the Runge-Kutta method.

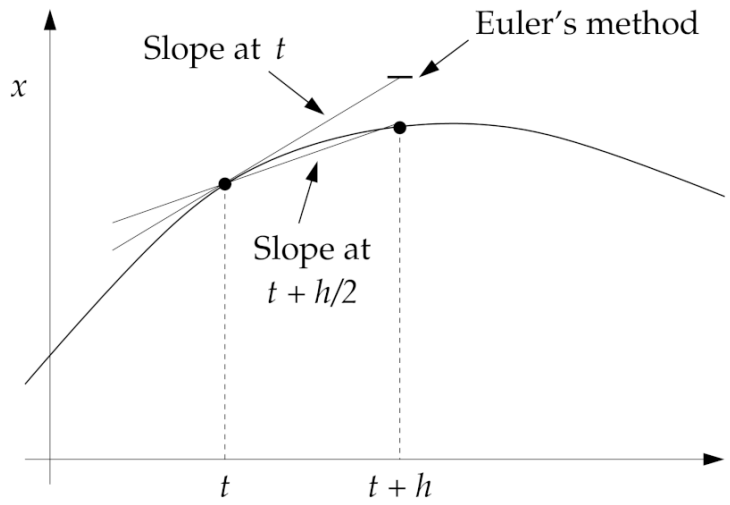

Figure 1: Euler's method and the second-order Runge-Kutta method that uses the slope at the midpoint to estimate $x(t + h)$.

The Runge-Kutta method

To improve Euler's method, we can keep the second-order term in the Taylor expansion (Equation 4), which can be expressed as

$$\tfrac{1}{2}h^2\frac{d^2x}{dt^2} = \tfrac{1}{2}h^2\frac{df}{dt}. \tag{8}$$

However, we don't have an expression for df/dt because, in general, f is the product of a routine, not an analytical solution.

The Runge-Kutta method is a general method for solving ODEs to some order h^n, which n is some natural number. Technically, Euler's method is the first-order Runge-Kunge method. If we take the second-order method, which is sometimes called the *midpoint method*, we will have to estimate the second-order derivative of x. The solution of ODEs is illustrated in Figure 1, which Euler's method takes the derivative at $t = t$, but see that we can better estimate the value of $x(t + h)$ by using the derivative at the midpoint, $t + h/2$.

We can estimate the value of $x(t + h)$ by taking the Taylor expansion of both $x(t)$ and $x(t + h)$ around the midpoint $t + h/2$

$$x(t+h) = x(t+\tfrac{1}{2}h) + \tfrac{1}{2}h\left(\frac{dx}{dt}\right)_{t+h/2} + \tfrac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + O(h^3) \tag{9}$$

$$x(t) = x(t+\tfrac{1}{2}h) - \tfrac{1}{2}h\left(\frac{dx}{dt}\right)_{t+h/2} + \tfrac{1}{8}h^2\left(\frac{d^2x}{dt^2}\right)_{t+h/2} + O(h^3) \tag{10}$$

Subtracting the second expression from the first and rearranging gives

$$x(t+h) = x(t) + h\left(\frac{dx}{dt}\right)_{t+h/2} + O(h^3) \tag{11}$$

$$= x(t) + hf[x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h] + O(h^3) \tag{12}$$

Notice that the h^2 terms has completely vanished, and now the approximation has errors $O(h^3)$, which is a whole factor of h more accurate than Euler's method.

However, we have a predicament of calculating the slope at the midpoint. How do we compute this without shrinking the step-size? We can calculate it by using Euler's method,

$$x(t+\tfrac{1}{2}h) = x(t)\tfrac{1}{2}hf(x,t) \tag{13}$$

and then substituting this value into Equation (11). The complete calculation can be encapsulated into three equations:

$$k_1 = hf(x,t), \tag{14}$$

$$k_2 = hf(x+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h), \tag{15}$$

$$x(t+h) = x(t) + k_2. \tag{16}$$

This is the **second-order Runge-Kutta method**, and it's named so because it's accurate to h^2. It can be shown (see the book) that the Euler method to calculate the midpoint slope is accurate to order h^2, so it doesn't introduce any additional errors.

Example 8.2: The second-order Runge-Kutta method

We can take our code from the previous example and easily modify it to use the second-order Runge-Kutta method. Here we will see that it achieves a similar solution with only 50 steps.

```
from math import sin
import numpy as np
import matplotlib.pyplot as plt

def f(x,t):
    return -x**3 + sin(t)

a = 0.0           # Start of the interval
b = 10.0          # End of the interval
N = 1000          # Number of steps
h = (b-a)/N       # Size of a single step
x = 0.0           # Initial condition

tpoints = np.arange(a,b,h)
```

```
xpoints = np.zeros(N)
xpoints[0] = x
#
# To be completed in class: Loop over points to calculate x
#

plt.plot(tpoints,xpoints)
plt.xlabel("t")
plt.ylabel("x(t)")
plt.show()
```

Fourth-order Runge-Kutta method

We can take even higher-order terms in the Taylor expansion (Equation 4) to devise higher-order accurate Runge-Kutta methods. The "sweet spot" is fourth-order, which gives very high accuracy without much computing difficulty. It is probably the most widely used method to solve ODEs. Now we have five equations to solve for the next timestep:

$$k_1 = hf(x,t), \tag{17}$$
$$k_2 = hf(x+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h), \tag{18}$$
$$k_3 = hf(x+\tfrac{1}{2}k_2, t+\tfrac{1}{2}h), \tag{19}$$
$$k_4 = hf(x+k_3, t+h), \tag{20}$$
$$x(t+h) = x(t) + \tfrac{1}{6}(k_1+2k_2+2k_3+k_4). \tag{21}$$

The two previous examples can be modified to use the fourth-order Runge-Kutta method, and it approaches a suitable solution with only $N = 20$ points, whereas 1000 points were needed for Euler's method!

Solutions over infinite ranges

Just like numerically integrating over an infinite range, we can re-express an ODE in a change of variables to avoid integrating out to $t = \infty$. Let's define

$$u = \frac{t}{1+t}, \quad \text{or equivalently} \quad t = \frac{u}{1-u}, \tag{22}$$

so that $u \to 1$ as $t \to \infty$. Now we can use the chain rule to write

$$\frac{dx}{du}\frac{du}{dt} = f(x,t) \tag{23}$$

$$\frac{dx}{du} = \frac{dt}{du} f\left(x, \frac{u}{1-u}\right). \tag{24}$$

Using $dt/du = 1/(1-u)^2$, we can write

$$\frac{dx}{du} = \frac{1}{(1-u)^2} f\left(x, \frac{u}{1-u}\right). \tag{25}$$

Desiring an ODE in the form $dx/du = g(x,u)$, we define

$$g(x,u) = \frac{1}{(1-u)^2} f\left(x, \frac{u}{1-u}\right). \tag{26}$$

We can now solve x as a function of u and then remap the solution back to t for a solution of $x(t)$.

8.2: Diff. Equations with More than One Variable

Many physics problems have more than one variable, and the equations determining their evolution is a set of *simultaneous differential equations*. That is, each variable depends on any or all of the variables, as well as the independent variable t. For example,

$$\frac{dx}{dt} = xy - x, \qquad \frac{dy}{dt} = y - xy + \sin^2 \omega t. \tag{27}$$

Note that there is still only one independent variable t. A general form of two first-order simultaneous ODEs is

$$\frac{dx}{dt} = f_x(x,y,t), \qquad \frac{dy}{dt} = f_y(x,y,t), \tag{28}$$

where f_x and f_y are any type of function. We can generalize this system to an arbitrary number of dependent variables by putting the variables and functions into vectors: $\mathbf{r} = (x, y, \ldots)$ and $\mathbf{f}(\mathbf{r},t) = (f_x(\mathbf{r},t), f_y(\mathbf{r},t), \ldots)$:

$$\boxed{\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r},t)} \tag{29}$$

It is straightforward to solve these because Euler's method and the Runge-Kutta method also applies to a system of simultaneous ODEs. For example, the equations for the fourth-order Runge-Kutta method would read as:

$$\mathbf{k}_1 = h\mathbf{f}(\mathbf{r},t), \tag{30}$$
$$\mathbf{k}_2 = h\mathbf{f}(\mathbf{r} + \tfrac{1}{2}\mathbf{k}_1, t + \tfrac{1}{2}h), \tag{31}$$
$$\mathbf{k}_3 = h\mathbf{f}(\mathbf{r} + \tfrac{1}{2}\mathbf{k}_2, t + \tfrac{1}{2}h), \tag{32}$$
$$\mathbf{k}_4 = h\mathbf{f}(\mathbf{r} + \mathbf{k}_3, t + h), \tag{33}$$
$$\mathbf{r}(t+h) = \mathbf{r}(t) + \tfrac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \tag{34}$$

Note that this can solved with `numpy`'s vector notation and can be generalized for an arbitrarily large set of equations, as shown in the following example that solves Equation (27) from $t = 0 \to 10$ with $\omega = 1$ and initial condition of $x = y = 1$.

Example code: Simulataneous ODEs

```
import numpy as np
import matplotlib.pyplot as plt

def f(r,t):
    x = r[0]
    y = r[1]
    fx = x*y - x
    fy = y - x*y + np.sin(t)**2
    return np.array([fx,fy],float)

a = 0.0
b = 10.0
N = 1000
h = (b-a)/N

tpoints = np.arange(a,b,h)
r = np.zeros((N,2))
r[0] = [1.0, 1.0]

# To be completed in class: loop over time to solve for x and y, which
# will be stored in r[]

plt.plot(tpoints, r[:,0], label='x')
plt.plot(tpoints, r[:,1], label='y')
plt.xlabel("t")
plt.ylabel("x,y")
plt.show()
```

8.3: 2nd-order Differential Equations

In the last section, we learned how to solve first-order differential equations with multiple variables. They are pretty rare in physics, and second-order and third-order equations are more commonplace. Luckily, solving these equations numerically is just the following extension of solving first-order ODEs.

Let's first consider the simple case where we only have one dependent variable t with a second-order ODE:

$$\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right), \tag{35}$$

meaning that the second derivative can be any arbitrary function, including non-linear ones.

For example, what if we had the following ODE

$$\frac{d^2x}{dt^2} = \frac{1}{x}\left(\frac{dx}{dt}\right)^2 + 2\frac{dx}{dt} - x^2e^{-4t}. \tag{36}$$

We can extend this to the notation in the first-order methods by defining

$$y \equiv \frac{dx}{dt}, \tag{37}$$

leading to

$$\frac{dy}{dt} = f(x, y, t) \tag{38}$$

that is *exactly the same* as a first-order ODE, and we just learned how to solve them. Similar tricks can be used with higher-order ODEs.

$$\frac{d^3x}{dt^3} = f\left(x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, t\right) \tag{39}$$

A third-order ODE above will have two additional independent variables that are the first and second derivatives, $y \equiv dx/dt$ and $z \equiv dy/dt$, leading to a system of three first-order ODEs,

$$\frac{dz}{dt} = f(x, y, z, t). \tag{40}$$

Generalizing this to vector form, we can write

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{f}\left(\mathbf{r}, \frac{d\mathbf{r}}{dt}, t\right), \tag{41}$$

which is equvalent to the first-order equations

$$\frac{d\mathbf{r}}{dt} = \mathbf{s}, \qquad \frac{d\mathbf{s}}{dt} = \mathbf{f}(\mathbf{r}, \mathbf{s}, t). \tag{42}$$

Given an initial system of n equations of m^{th} order, a set of $m \times n$ simultaneous first-order equations will arise that can be solved with conventional methods.

Example 8.6: The non-linear pendulum

A standard problem in physics is the linear pendulum, where you approximate the behavior of a pendulum by a linear differential equation that can be solved exactly. But a real pendulum is non-linear. Consider a pendulum with an arm of length l holding a ball of mass m. In terms of the angle θ of displacement of the arm from the vertical, the acceleration of the mass is $l(d^2\theta/dt^2)$ in the tangential direction. Meanwhile the force on the mass is vertically downward with magnitude mg, where $g = 9.81$ m/s^2, and for simplicity, we ignore friction and the mass of the arm. The component of this force in the tangential direction is $mg\sin\theta$,

always toward the rest point $\theta = 0$, and hence Newton's second law gives us an equation of motion for the pendulum of the form

$$ml\frac{d^2\theta}{dt^2} = -mg\sin\theta \tag{43}$$

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta \tag{44}$$

Let's rewrite this second-order ODE as two first-order ones.

$$\frac{d\theta}{dt} = \omega, \qquad \frac{d\omega}{dt} = -\frac{g}{l}\sin\theta \tag{45}$$

Between them, these two first-order ODEs are equivalent to the one second-order ODE. Now we combine the two variables θ and ω into a single vector $\mathbf{r} = (\theta, \omega)$ and apply the fourth-order Runge-Kutta method in vector form to solve the two equations simulataneously. We are only really interested in the solution for one of the variables, θ. The method gives us the solution for both, but we can simply ignore the value of ω if we don't need it. The program will be similar to that of Example 8.5, except that the function `f(r,t)` must be redefined appropriately. If the arm of the pendulum were $l = 10$ cm and release the pendulum from rest at $\theta = 179°$, we would have

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

g = 9.81  # gravity
l = 0.1   # arm length of pendulum [m]
theta0 = 179.0 # initial arm position [degrees]

a = 0.0   # start time
b = 10.0  # end time
N = 1000
h = (b-a)/N

# In-class problem: define function for an equivalent set of two
# first-order equations
def f(r,t):
    return [0,0]


tpoints = np.arange(a,b,h)
r = np.zeros((N,2))
r[0] = [np.pi * theta0/180.0, 0.0]

# Loop over time to solve for x and y, which will be stored in r[]
```

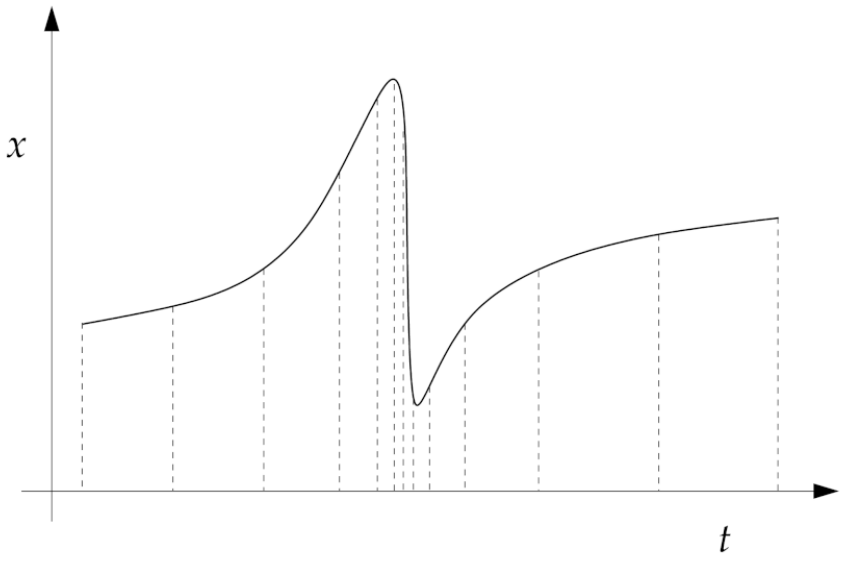

Figure 2: Adaptive step sizes

```
for i,t in enumerate(tpoints[1:],1):
    k1 = h * f(r[i-1], t)
    k2 = h * f(r[i-1]+0.5*k1, t+0.5*h)
    k3 = h * f(r[i-1]+0.5*k2, t+0.5*h)
    k4 = h * f(r[i-1]+k3, t+h)
    r[i] = r[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6.0

plt.plot(tpoints, r[:,0], label=r'$\theta$')
plt.plot(tpoints, r[:,1], label=r'$\omega$')
plt.xlabel("t")
plt.ylabel(r"$\theta, \omega$")
plt.legend()
plt.show()
```

8.4: Varying the Step Size

So far, the methods have used a fixed step size that is user-defined. But we can achieve more accuracy results if the step size h is allowed to vary during the calculation. Suppose if we're solving a first-order ODE $dx/dt = f(x,t)$ that slowly varies in some regions but quickly varies in others, illustrated in the Figure. In the latter region, we need small step sizes to limit the amount of error, which is dependent on the steepness of the slope. On the other hand, the calculation can still be accurate in relatively flat regions with large step sizes.

The idea behind an **adaptive step size** method is to vary the step sizes h so that the error introduced per unit interval t is roughly constant. In practice, this method has two

parts. First we have to estimate the error of our steps, then we compare that error to our required accuracy and either increase or decrease the step size to achieve the desired accuracy.

We will use the fourth-order Runge-Kutta method as an example. First we choose some initial step h, and take two time steps to bring us to time $t + 2h$. Now we go back to the previous time step and take a single step size $2h$. We compare the two results of $x(t + 2h)$ to test the accuracy of our calculation. We know that this fourth-order method has errors of fifth-order, ch^5 with c as some constant, for a single timestep. So for two steps, the true value of $x(t + 2h)$ is related to our estimated (calculated) value x_1 by

$$x(t + 2h) = x_1 + 2ch^5. \tag{46}$$

When we perform a single larger timestep, the error is $c(2h)^5 = 32ch^5$, giving

$$x(t + 2h) = x_2 + 32ch^5, \tag{47}$$

where x_2 is our calculated value with a stepsize $2h$. We can equate these two solutions and find that the error per step is

$$\epsilon \equiv ch^5 = \tfrac{1}{30}(x_1 - x_2) \tag{48}$$

Our goal is to make this error exactly equal to some target user-defined accuracy. If ϵ is smaller than the target, then the stepsize can be enlarged; if it is larger than the target, then this is unacceptable and the calculation needs to be redone.

Let's try to calculate the stepsize that results in an error *exactly* to the target accuracy δ per unit time (but not more). This "perfect" stepsize is called h'. If we were to take steps of size h' then the error on a single step would be

$$\epsilon' = ch'^5 = ch^5 \left(\frac{h'}{h}\right)^5 = \tfrac{1}{30}(x_1 - x_2) \left(\frac{h'}{h}\right)^5 \tag{49}$$

We next set this error equal to the target accuracy in a single step,

$$\tfrac{1}{30}|x_1 - x_2| \left(\frac{h'}{h}\right)^5 = h'\delta. \tag{50}$$

Solving for h', we find

$$h' = h \left(\frac{30h\delta}{|x_1 - x_2|}\right)^{1/4} = h\rho^{1/4}, \tag{51}$$

where $\rho \equiv 30h\delta/|x_1 - x_2|$.

The complete **adaptive step size method** is as follows.

1. Perform two steps of size h and then from the same starting point, one step of size $2h$.
2. From these two calculations, compute ρ from the two estimates x_1 and x_2 at time $t + 2h$.

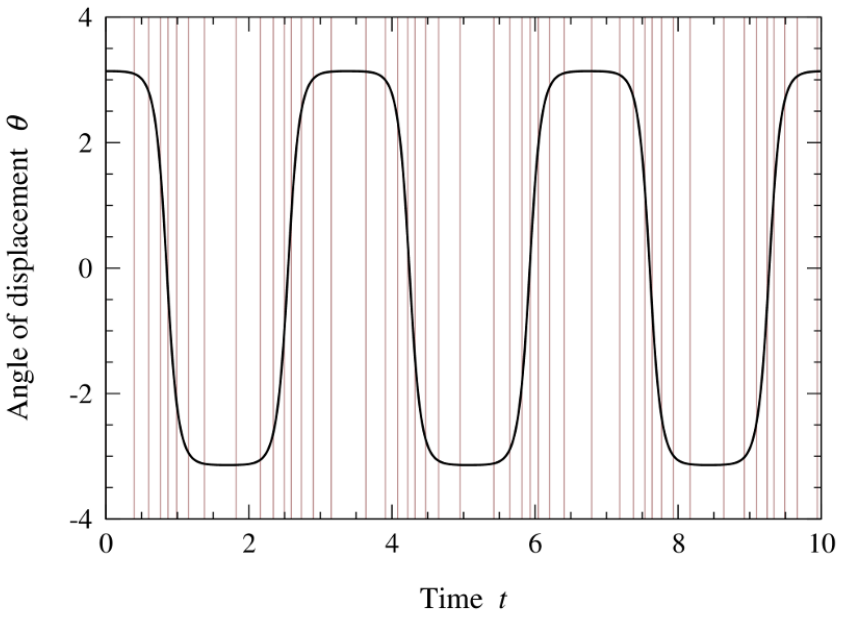

Figure 3: **Motion of a non-linear pendulum.** Displacement angle as a function of time. The vertical lines show every 20^{th} timestep in an adaptive Runge-Kutta method.

3. If $\rho > 1$, then the actual accuracy is better than the target accuracy. We keep the x_1 estimate (2 steps). To avoid computational waste in the next step, we increase h by a factor determined from Equation (51).
4. If $\rho < 1$, then the actual accuracy is poorer than the target accuracy. We have to repeat the calculation with a smaller timestep determined from Equation (51).
5. *Note:* Usually the step size is not allowed to change more than a factor of two.

Each step requires more work for the adaptive method than the fixed timestep method. This extra effort pays off because we know that the solution is better than some specified accuracy. It is even more accurate than fixed timestep methods because numerical errors are culminative in the methods discussed so far, and it prevents them from getting out of hand in "problem areas" with steep gradients in the solution. One good example of steep gradients is an object in orbit around the Sun, such as a comet, as it makes its closest approach to the Sun, called the perihelion. The adaptive method is illustrated in Figure 3 for the non-linear pendulum problem. Usually, the adaptive method requires much fewer timesteps and thus less overall time.

One interesting and useful idea in adaptive methods relates the "double-step" and single-step together for a more accurate solution. Recall that the single-step estimate x_1 is related to the actual solution by

$$x(t + 2h) = x_1 + 2ch^5 + O(h^6), \tag{52}$$

but we have also seen that the two estimates are related by

$$ch^5 = \tfrac{1}{30}(x_1 - x_2). \tag{53}$$

We can plug this into Equation (52),

$$x(t+2h) = x_1 + \tfrac{1}{15}(x_1 - x_2) + O(h^6), \tag{54}$$

and now we have a fifth-order accurate method! This is called **local extrapolation**, and it's kind of a free bonus prize that comes along with the adaptive method. The only downside is that we don't know the magnitude of the error, but it is presumably better than the fourth-order method, providing a modest improvement on Runge-Kutta methods.

The real interesting part of this addition is the extrapolation, namely *Richardson extrapolation*. Can we take this idea of further correcting the errors in an iterative fashion, making the order of accuracy higher and higher? This is exactly the same idea that led to Romberg integration. We will explore this in more detail next.

8.5: Other Methods for Differential Equations

The widely used Runge-Kutta method provides a straightforward and robust scheme to solve ODEs. However there are some situations in which it is not optimal, for instance if energy conservation is desired. In this section, we will cover the leapfrog, Verlet, Bulirsch-Stoer methods.

The Leapfrog method

Consider the first-order ODE in a single variable:

$$\frac{dx}{dt} = f(x,t) \tag{55}$$

Previously, we introduced the second-order Runge-Kutta method, also known as the midpoint method, that estimated the next timestep $x(t+h)$ using the slope at the midpoint, $f(x(t+h/2), t+h/2)$. Because the slope at the midpoint is usually unknown, multiple steps were needed:

$$x(t+\tfrac{1}{2}h) = x(t) + \tfrac{1}{2}f(x,t) \tag{56}$$

$$x(t+h) = x(t) + hf(x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h). \tag{57}$$

It is second-order accurate, but with repeated steps, the errors accumulate, making the total error first-order accurate. The top panel in Figure 4 demonstrates how this method is calculating each timestep.

The leapfrog method is a variant of the midpoint method to avoid taking half timesteps for the slope in every calculation. After an initial half timestep, it advances the first derivative by a full timestep (see the bottom panel of Figure 4),

$$x(t+\tfrac{3}{2}h) = x(t+\tfrac{1}{2}h) + hf(x(t+h), t+h). \tag{58}$$

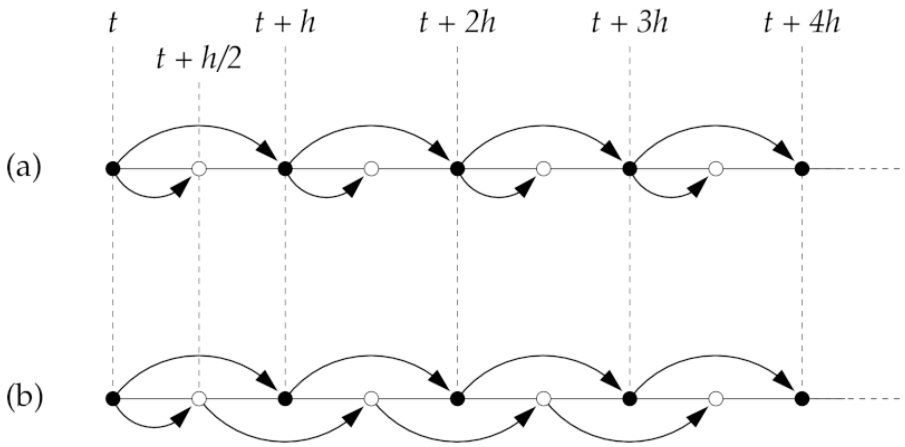

Figure 4: Illustration of the time-stepping of the second-order Runge-Kutta (top) and the Leapfrog method (bottom)

In this calculation, the gradient at the integer timestep is the midpoint of the half timestep step. Instead of being first-order accurate with the estimate using the Euler method, it is still second-order accurate. Then we can use this half timestep at $t + \frac{3}{2}h$ for the next full step:

$$x(t + 2h) = x(t + h) + hf(x(t + \tfrac{3}{2}h), t + \tfrac{3}{2}h). \tag{59}$$

This method is called the leapfrog method because it "leaps over" the position of the previously calculated value. It is still second-order accurate, so that it is accurate to order h over time and its total accumulated error has an order h^2, just like the second-order Runge-Kutta method. However, it carries the special property of conserving energy over the integration, which we will show later.

Lastly just like other methods, it can be easily extended to simultaneous ODEs:

$$\mathbf{r}(t + h) = \mathbf{r}(t) + h\mathbf{f}(\mathbf{r}(t + \tfrac{1}{2}h), t + \tfrac{1}{2}h) \tag{60}$$

$$\mathbf{r}(t + \tfrac{3}{2}h) = \mathbf{r}(t + \tfrac{1}{2}h) + h\mathbf{f}(\mathbf{r}(t + h), t + h) \tag{61}$$

Time reversal and energy conservation

The leapfrog method is time-reversal symmetric, which implies that it conserves energy throughout the calculation. Let's consider the solution of $x(t)$. At some time t_1, we need two values to propagate the solution: $x(t_1)$ and $x(t_1 + \frac{1}{2}h)$. Given only these values, we can move forward with the solution. Suppose that we continue the calculation until some later time t_2, where we have $x(t_2)$ and $x(t_2 + \frac{1}{2}h)$, and we want to go backwards in time to recover the original solution at time t_1. We set the time interval to $-h$ to achieve this reversal.

Changing $t \to -h$ in the leapfrog method, we find that

$$x(t-h) = x(t) - hf(x(t-\tfrac{1}{2}h), t-\tfrac{1}{2}h) \tag{62}$$
$$x(t-\tfrac{3}{2}h) = x(t-\tfrac{1}{2}h) - hf(x(t-h), t-h). \tag{63}$$

Now to recover $x(t)$, we set $t \to \frac{3}{2}h$,

$$x(t+\tfrac{1}{2}h) = x(t+\tfrac{3}{2}h) - hf(x(t+h), t+h) \tag{64}$$
$$x(t) = x(t+h) - hf(x(t+\tfrac{1}{2}h), t+\tfrac{1}{2}h), \tag{65}$$

which is exactly the same formulation of $x(t+h)$ but in reverse.

This **is not true** for the Runge-Kutta method that will give differing solutions in either the forward or backward directions, even if you correct for rounding and approximation errors. For reference, here are the governing equations for the second-order Runge-Kutta method, which the reader can check that it is not time-reversible.

$$k_1 = hf(x,t), \tag{66}$$
$$k_2 = hf(x+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h), \tag{67}$$
$$x(t+h) = x(t) + k_2. \tag{68}$$

To demonstrate the energy conservation of the leapfrog method, consider a frictionless nonlinear pendulum, described by

$$\frac{d\theta}{dt} = \omega, \quad \frac{d\omega}{dt} = -\frac{g}{l}\sin\theta. \tag{69}$$

We previously saw that the second-order Runge-Kutta method gave a pretty good solution. This ideal pendulum should have constant total energy, and we can easily calculate the total energy in every timestep, shown in the top panel of Figure 5.

Comparing the total energy to the solution given by the leapfrog method, one can see that the total energy is indeed smaller for Runge-Kutta, but it is steadily increasing with time. This is compared to the leapfrog method that has larger overall fluctuations, but its time-averaged total energy is constant. Over time, the pendulum will gain energy with the Runge-Kutta method, eventually diverging from the actual solution. Because the leapfrog method is time-reversible, its total energy will always return to the initial value at the starting displacement angle.

In general, the leapfrog method should be used for any periodic system, such as a pendulum or orbital systems, because it will conserve energy over long periods of time (many periods), providing a stable solution.

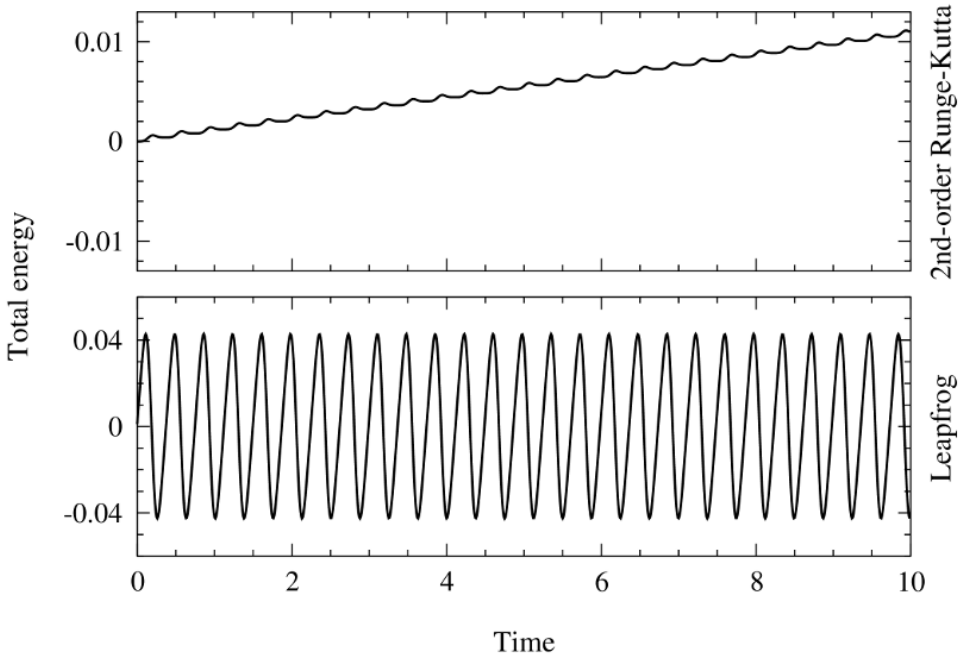

Figure 5: Total energy of the non-linear pendulum using the second-order Runge-Kutta (top) and leapfrog (bottom) methods.

Verlet method

Suppose that we are using the leapfrog method to solve a second-order differential equation, such as Newton's second law,

$$\frac{d^2x}{dt^2} = f(x,t), \tag{70}$$

which can be written as a set of two coupled first-order ODEs,

$$\frac{dx}{dt} = v, \quad \frac{dv}{dt} = f(x,t). \tag{71}$$

By defining the vector $\mathbf{r} = (r, v)$, we can write it as a single equation,

$$\frac{d\mathbf{r}}{dt} = \mathbf{f}(\mathbf{r}, t). \tag{72}$$

Let's inspect the numerical system further by writing out the full description of the leapfrog method,

$$x(t+h) = x(t) + hv(t + \tfrac{1}{2}h) \tag{73}$$

with the value of v at the next midpoint

$$v(t + \tfrac{3}{2}h) = v(t + \tfrac{1}{2}h) + hf(x(t+h), t+h). \tag{74}$$

Notice that we never have to calculate v at full timesteps. This is a special case, where (1) the right-hand side of the first equation only depends on v and not x, and (2) the right-hand side of the second equation only depends on x and not v. Many physics problems have this form.

The only downside to this method occurs when we want to calculate some quantity that depends on v at an integer timestep, such as total energy. But there's an easy solution to this problem. If we did know $v(t+h)$, we can use Euler's method with a backwards half timestep to calculate $v(t+\frac{1}{2}h)$,

$$v(t+\tfrac{1}{2}h) = v(t+h) - \tfrac{1}{2}hf(x(t+h), t+h), \tag{75}$$

which can be rearranged as

$$v(t+h) = v(t+\tfrac{1}{2}h) + \tfrac{1}{2}hf(x(t+h), t+h), \tag{76}$$

giving the values of v at integer steps in quantities that we already know from the leapfrog method. Putting it all together, the **Verlet method** is described by the following equations, given initial values of x and v. First, we need to calculate the velocity at the first half timestep,

$$v(t+\tfrac{1}{2}h) = v(t) + \tfrac{1}{2}hf(x(t), t). \tag{77}$$

Then the subsequent values of x and v are calculated by

$$x(t+h) = x(t) + hv(t+\tfrac{1}{2}h) \tag{78}$$
$$k = hf(x(t+h), t+h) \tag{79}$$
$$v(t+h) = v(t+\tfrac{1}{2}h) + \tfrac{1}{2}k \tag{80}$$
$$v(t+\tfrac{3}{2}h) = v(t+\tfrac{1}{2}h) + k \tag{81}$$

As with other ODE solvers, it can extended to any simultaneous set of ODEs by replacing $\mathbf{r} = (x, y, \ldots)$, $\mathbf{v} = d\mathbf{r}/dt$, and $d^2\mathbf{r}/dt^2 = \mathbf{f}(\mathbf{r}, t)$.

The modified midpoint method

The leapfrog method has the advantage over the Runge-Kutta method because it is time reversible **and** its errors are an even power of h with its truncation error being a sum of only even powers and no odd terms. More generally, we can write the error as some function of h, $\epsilon(h)$. Because it has the nice property of time reversibility, the error going backwards in time is just the negative of the error going forward in time,

$$\epsilon(-h) = -\epsilon(h), \tag{82}$$

telling us that ϵ is an odd function. We can expand the error as such a function,

$$\epsilon(h) = c_3h^3 + c_5h^5 + c_7h^7 + \ldots, \tag{83}$$

where c_i are constants. We also need to consider that this error is for a single timestep, whereas the error is compounded by an order of h over many timesteps.

The error equation above is correct with one catch. To perform the initial step, we need to use Euler's method to calculate the slope at the first midpoint. Because Euler's method has errors on the order of h^2, this spoils the errors that are associated with all subsequent steps. But there is a workaround to this fault!

Imagine that we want to integrate some ODE to a much later time $t + H$, using n steps of size H/n. Let's write the leapfrog method in a slightly different form.

$$x_0 = x(t), \tag{84}$$
$$y_1 = x_0 + \tfrac{1}{2}hf(x_0, t). \tag{85}$$

Then we follow up this initial step with

$$x_1 = x_0 + hf(y_1, t + \tfrac{1}{2}h), \tag{86}$$
$$y_2 = y_1 + hf(x_1, t + h), \tag{87}$$
$$x_2 = x_1 + hf(y_2, t + \tfrac{3}{2}h). \tag{88}$$

The leapfrog method has the general form where we need the solution x_m at integer multiples of h and values of y_m at half-integer multiples, formulated as the following

$$y_{m+1} = y_m + hf(x_m, t + mh) \tag{89}$$
$$x_{m+1} = x_m + hf(y_{m+1}, t + (m + \tfrac{1}{2})h). \tag{90}$$

To finalize the solution, we need the last two points to arrive at time $t+H$: $y_n = x(t+H-\frac{1}{2}h)$ and $x_n = x(t + H)$. The easiest solution would be to take x_n as our final solution. However, we can use the last midpoint result to calculate the final point $x(t+H)$, using Euler's method,

$$x(t + H) = y_n + \tfrac{1}{2}hf(x_n, t + H). \tag{91}$$

So we have two ways to calculate the final point. We can take the average of both methods and arrive at

$$x(t + H) = \tfrac{1}{2}[x_n + y_n + \tfrac{1}{2}hf(x_n, t + H)], \tag{92}$$

which amazingly cancels out the errors that were originally introduced in the first midpoint calculation, using Euler's method. This restores the error's complete depends on *only even* powers of h. This is known as the **modified midpoint method**, but it isn't widely used because it gives very little benefits over the plain leapfrog method and is less accurate than the fourth-order Runge-Kutta method. However by doing this correction at the end, it gives us some insight on the next method we will study, the Bulirsch-Stoer method.

The Bulirsch-Stoer method

This method combines two concepts that we've studied before: the modified midpoint method and Richardson extrapolation. It's similar to how Romberg integration work, where the errors are continually corrected as the stepsize is decreased.

First let's take a first-order ODE as a working example:

$$\frac{dx}{dt} = f(x,t) \tag{93}$$

that will be solved from an initial time t to a later time $t+H$. We start by using the modified midpoint method, but only using a single timestep of size $h_1 = H$. This gives an estimate of the value $x(t+H)$, which we will call $R_{1,1}$. If H is a large interval, then $R_{1,1}$ will be a crude estimate.

Next we go back to the initial time and take timesteps of size $h_2 = H/2$, giving us a second estimate, which we call $R_{2,1}$. We showed previously that the modified midpoint method has an error with only even terms, so that the actual solution can be expressed as

$$x(t+H) = R_{2,1} + c_1 h_2^2 + O(h_2^4) \tag{94}$$

and similarly we can use the first estimate,

$$x(t+H) = R_{1,1} + c_1 h_1^2 + O(h_1^4) = R_{1,1} + 4c_1 h_2^2 + O(h_2^4), \tag{95}$$

where we have used $h_1 = 2h_2$ and c_i is some constant. Equating these two expressions, we find that the error term is

$$c_1 h_2^2 = \tfrac{1}{3}(R_{2,1} - R_{1,1}) \tag{96}$$

that can be substituted back into Equation (94) to find

$$t(t+H) = R_{2,1} + \tfrac{1}{3}(R_{2,1} - R_{1,1}) + O(h_2^4), \tag{97}$$

that has fourth-order errors. We can keep on going, decreasing the stepsize and using the error estimate to "correct" our coarse solution. Now decreasing the stepsize to $h_3 = H/3$, we find

$$R_{3,2} = R_{3,1} + \tfrac{4}{5}(R_{3,1} - R_{2,1}) \tag{98}$$

that has errors of order h_3^4. We can now write our unrefined estimate as

$$x(t+H) = R_{3,2} + c_2 h_3^4 + O(h_3^6). \tag{99}$$

We also have the estimate from the previous step,

$$x(t+H) = R_{2,2} + c_2 h_2^4 + O(h_2^6) = R_{2,2} + \tfrac{81}{16} c_2 h_3^4 + O(h_3^6), \tag{100}$$

where we've used $h_2 = \frac{3}{2}h_3$. Equating these two expressions, we can solve for the error term to find

$$c_2 h_3^4 = \tfrac{16}{65}(R_{3,2} - R_{2,2}). \tag{101}$$

We can plug this into Equation (99) to arrive at

$$x(t+H) = R_{3,3} + O(h_3^6) \tag{102}$$

$$R_{3,3} = R_{3,2} + \tfrac{16}{65}(R_{3,2} - R_{2,2}). \tag{103}$$

We are already at errors that are order h^6 with only three midpoint steps. We continue to an arbitrary number of refinements, and the general formalism for the Bulirsch-Stoer method is

$$x(t+H) = R_{n,m} + c_m h_n^{2m} + O(h_n^{2m+2}), \tag{104}$$

where n represents the number of steps taken so far, and m is the m^{th} estimate. For example, $m = 1$ is the original modified midpoint method, and the $m > 1$ terms are given by Richardson extrapolation. The corresponding estimate $R_{n-1,m}$ made with one less step satisfies

$$t(x+H) = R_{n-1,m} + c_m h_{n-1}^{2m} + O(h_n^{2m+2}). \tag{105}$$

We can use the fact that $h_n = H/n$ and $h_{n-1} = H/(n-1)$ to equate the previous two equations and solve for the error term:

$$c_m h_n^{2m} = \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1}. \tag{106}$$

We substitute this term into Equation (104) to obtain the next estimate

$$x(t+H) = R_{n,m+1} + O(h_n^{2m+2}) \tag{107}$$

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1}. \tag{108}$$

This is the heart of Richardson extrapolation and thus the Bulirsch-Stoer solution. It allows us to make very accurate estimates with few timesteps because we gradually eliminate the truncation (approximation) error.

For each value of n, we calculate a basic modified midpoint estimate $R_{n,1}$ with n steps, and then a series of further extrapolation estimates. Each extrapolation estimate depends on two previous estimates, as and the last estimate in each series ($m = n$) is the highest-order estimate for that value of n. Also at each step, we have an error measure $c_m h_n^{2m}$ (Equation 106) that can be compared with a user-defined target accuracy. We continue dividing up the solution (i.e. increasing n) until the target is reached.

There are some downsides to the Bulirsch-Stoer method, though. First, it only gives an accurate solution at time $t+H$, and at the intermediate points, we only have the raw modified method estimates, which are not comparatively accurate. Second, this method is only useful **if the solution converges within 8–10 steps**. It requires too much computational work if hundreds or thousands of terms are needed. The workaround to both of these problems is to subdivide the solution's total interval ($t = a \to b$) into N portions of size $H = (b-a)/N$. Then we apply the Bulirsch-Stoer method to each division, obtaining a very accurate solution at each subdivision boundary.

In summary, here is the complete method. Let δ be the target accuracy per unit time. Divide the entire solution into N equal intervals of length H each, and apply the following steps:

1. Set $n = 1$ and use the modified midpoint method to calculate $R_{1,1}$ of the solution from $t \to t + H$ using just *one step*.
2. Increase n by one and calculate a new modified midpoint estimate $R_{n,1}$ with that many steps.
3. Use Equation (108) to calculate further estimate $R_{n,2} \cdots R_{n,n}$.
4. After calculating the whole set of estimates, compare the error given by Equation (106) with the target accuracy $H\delta$.
   (a) If the error is larger than the target accuracy, go back to step #2.
   (b) Otherwise, move on to the next time interval H.

Example 8.7: Bulirsch-Stoer method for the non-linear pendulum

Let's consider our non-linear pendulum again with a starting angle $\theta = 179°$ just like before. Its motion is described by the following two first-order ODEs,

$$\frac{d\theta}{dt} = \omega, \quad \frac{d\omega}{dt} = -\frac{g}{l}\sin\theta. \tag{109}$$

Here is a program that uses the Bulirsch-Stoer method to solve it. It is much more involved than any other method that we've studied so far.

```
#%matplotlib inline
from math import sin,pi
import numpy as np
import matplotlib.pyplot as plt

g = 9.81
l = 0.1
theta0 = 179*pi/180

a = 0.0
b = 10.0
N = 100          # Number of "big steps"
H = (b-a)/N      # Size of "big steps"
delta = 1e-8     # Required position accuracy per unit time

def f(r):
    theta = r[0]
    omega = r[1]
    ftheta = omega
```

```
    fomega = -(g/l)*sin(theta)
    return np.array([ftheta,fomega],float)

tpoints = np.arange(a,b,H)
thetapoints = []
r = np.array([theta0,0.0],float)

# Do the "big steps" of size H
for t in tpoints:

    thetapoints.append(r[0])

    # Do one modified midpoint step to get things started
    n = 1
    r1 = r + 0.5*H*f(r)
    r2 = r + H*f(r1)

    # The array R1 stores the first row of the
    # extrapolation table, which contains only the single
    # modified midpoint estimate of the solution at the
    # end of the interval
    R1 = np.empty([1,2],float)
    R1[0] = 0.5*(r1 + r2 + 0.5*H*f(r2))

    # Now increase n until the required accuracy is reached
    error = 2*H*delta
    while error>H*delta:

        n += 1
        h = H/n

        # Modified midpoint method
        r1 = r + 0.5*h*f(r)
        r2 = r + h*f(r1)
        for i in range(n-1):
            r1 += h*f(r2)
            r2 += h*f(r1)

        # Calculate extrapolation estimates.  Arrays R1 and R2
        # hold the two most recent lines of the table
        R2 = R1
        R1 = np.empty([n,2],float)
        R1[0] = 0.5*(r1 + r2 + 0.5*h*f(r2))
        for m in range(1,n):
            epsilon = (R1[m-1]-R2[m-1])/((n/(n-1.0))**(2*m)-1)
```

```
            R1[m] = R1[m-1] + epsilon
        error = abs(epsilon[0])

    # Set r equal to the most accurate estimate we have,
    # before moving on to the next big step
    r = R1[n-1]

# Plot the results
plt.plot(tpoints,thetapoints)
plt.plot(tpoints,thetapoints,"b.")
plt.show()
```

This method produces nearly the same result as the adaptive Runge-Kutta method, but with *fewer timesteps and less computational work*: 3,800 steps (including all raw intermediate midpoint estimates) versus 4,200 steps for adaptive Runge-Kutta. However for each adaptive Runge-Kutta step, we need to call the function `f(r)` four times compared to two in the Bulirsch-Stoer method. So the number of operations required for the solution would be approximately proportional to the 7,600 function calls for Bulirsch-Stoer versus 16,800 function calls for adaptive Runge-Kutta.

Interval size for the Bulirsch-Stoer method

In the example above, we chose the value for N manually, which works well for simple problems. However for more complex and demanding problems, we would like N to be calculated based on the problem at-hand. One method is to choose a small value for N, say 1, 2, or 4. Then we use the Bulirsch-Stoer method on the first section. If we don't reach the target accuracy by $n = 8 - 10$ (midpoint intervals), we throw away that solution, and divide this particular subdivision into two pieces with lengths of $\frac{1}{2}H$. Then we apply the same method on these smaller sections. We continue this iterative process repeatedly until we reach the target accuracy, most likely resulting in subdivisions that are unequally spaced through the time domain.