BRYN MAWR COLLEGE

Department of Physics

Computational Module 10 – Differential Equations

Prerequisite modules: Module 3 – Iterative Methods, Module 4 – Differentiation
Estimated completion time: 3-5 hours
Learning objectives: Become familiar with some common techniques for solving ordinary and partial differential equations

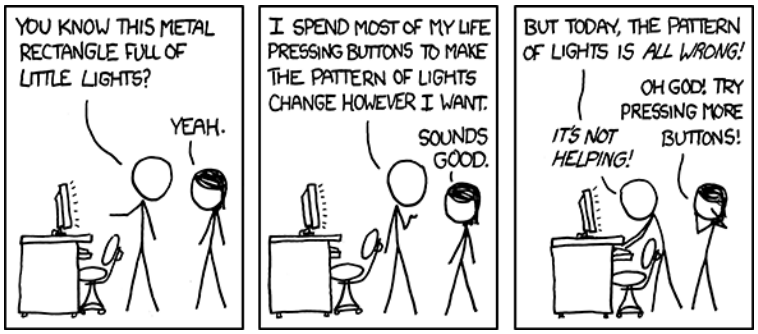


Image credit: xkcd.com

Ordinary differential equations (ODEs) are common in physics, and examples include Newton's second law, Schrödinger's equation in one dimension, and the one-dimensional wave equation. Even nonlinear ODEs, which usually cannot be solved analytically, can be handled numerically. The solution of differential equations is one of the most important applications of numerical methods. This module presents some of the key methods used to solve ordinary differential equations.[1] Partial differential equations will be considered in Module 11.

10.1 First-Order Equations of One Variable

We start our exploration by considering first-order differential equations of one variable. As a reminder, "first-order" in this context means that the equation involves no derivative higher than the first. Our model equation will be

[1] This Module adapted from *Computational Physics*, by Mark Newman.

$$\frac{dx(t)}{dt} = f(x,t), \tag{1}$$

where t is the independent variable and $x(t)$ is the dependent one. (So, to be pedantic, our "one variable" is x in this case; t can be considered a parameter.) Here, we're thinking of t as time and x as position, but they could just as well both be position coordinates, in which case the equation would have a form like $dy(x)/dx = f(y,x)$.

A full solution to Eq. (1) requires specification of an ***initial condition*** or ***boundary condition*** which, for this sort of equation, generally means the value $x(0)$, although specification at other times is perfectly fine.

10.1.1 Euler's Method

Perhaps the simplest approach to solving ODEs of one variable is ***Euler's Method***, introduced in Module 3 and based on the Taylor expansion of the function $x(t)$ about t using some small time interval h, as in

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

For small enough intervals h, we can ignore the terms of order h^2 and higher, leaving just

$$x(t+h) = x(t) + h\,\frac{dx}{dt} = x(t) + hf(x,t). \tag{3}$$

This is just like the kinematic equation $x(t+h) = x(t) + h\,dx/dt$ we used in Module 3. This more general context shows that in the earlier module it was important to make the time step (dt) small, so that higher-order terms (specifically, the second-derivative term associated with the acceleration) could be neglected. It turns out, though, that while the error at each time step is $\mathrm{O}(h^2)$, when summed over all of the time steps the total error becomes $\mathrm{O}(h)$. [This is because each step involves $\mathrm{O}(h^2)$ error, and the number of steps needed to span a given range is inversely proportional to h, so the overall error is $\mathrm{O}(h^2) \times 1/h = \mathrm{O}(h)$.] Thus, Euler's method is not very accurate, and since a much more accurate and equally fast alternative exists, Euler's method is not used for ODEs (although it does turn out to be useful in the solution of *partial* differential equations). We now turn to the more accurate method.

10.1.2 Runge-Kutta Method

An obvious improvement on Euler's method is to include higher-order terms in the Taylor expansion, but it turns out that this generally isn't a useful approach, since it requires knowing the derivative $dx^2/dt^2 = df/dt$, which can be determined only when the function f is known explicitly. Often, however, this function is the output of another program and does not have an analytic form. In

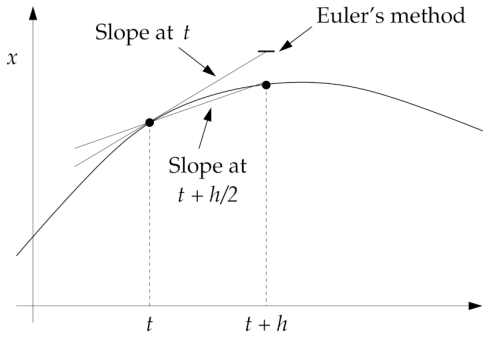


Figure 1: Comparison of first- and second-order Runge-Kutta methods

any case, the method to be presented here is more convenient, has better accuracy, and does not require the computation of any derivatives.

This wonderful alternative approach is known as the ***Runge-Kutta*** (RK) ***method***, after the two German mathematicians who developed it around 1900. In fact, it really is a set of approaches of different orders, of which Euler's method is just the first-order approach. The difference between Euler's method and the second-order RK (hereafter, "RK2") method is illustrated in Figure 1. Euler's method starts from the value of the function at time t, $x(t)$, and extrapolates the slope at that time, $f(x,t)$, to predict the value $x(t+h)$. The RK2 method instead uses the value of the slope at the *midpoint* $t + h/2$. One can see from the figure that the midpoint slope does a much better job of predicting the later value of the function than the slope at the earlier time t does. The two methods do equally well for a function that's a straight line but, generally speaking, the superiority of the second-order method is greater the more curved the function is in the time interval $t \to t+h$.

Performing a Taylor expansion about $t + h/2$ to obtain the value of $x(t + h)$, we have

$$x(t+h) = x(t+\tfrac{1}{2}h) + \tfrac{1}{2}h\left(\frac{dx}{dt}\right)_{t+\frac{1}{2}h} + \tfrac{1}{2!}\left(\tfrac{1}{2}h\right)^2\left(\frac{d^2x}{dt^2}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3). \tag{4}$$

We can write a similar expression for $x(t)$ in terms of $x(t + h/2)$:

$$x(t) = x(t+\tfrac{1}{2}h) + \left(-\tfrac{1}{2}h\right)\left(\frac{dx}{dt}\right)_{t+\frac{1}{2}h} + \tfrac{1}{2!}\left(-\tfrac{1}{2}h\right)^2\left(\frac{d^2x}{dt^2}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3). \tag{5}$$

Subtracting the second equation from the first one and rearranging terms yields

$$x(t+h) = x(t) + h\left(\frac{dx}{dt}\right)_{t+\frac{1}{2}h} + \mathrm{O}(h^3) \tag{6}$$

$$= x(t) + hf\left(x\left(t+\tfrac{1}{2}h\right), t+\tfrac{1}{2}h\right) + \mathrm{O}(h^3). \tag{7}$$

Notice that the $\mathrm{O}(h^2)$ term has disappeared, and that the error term in this case is $\mathrm{O}(h^3)$. Thus,

this approximation is a factor of h better in terms of accuracy than Euler's method.

There is one small problem in the application of this method — in general, we don't know $x(t+\frac{1}{2}h)$. It can, however, be approximated using Euler's method: $x(t+\frac{1}{2}h) \simeq x(t) + \frac{1}{2}hf(x,t)$. When this is substituted into the previous expression, we obtain

$$x(t+h) = x(t) + hf\left(x + \tfrac{1}{2}hf(x,t), t + \tfrac{1}{2}h\right). \tag{8}$$

Although you might suspect that this introduces an $\mathrm{O}(h^2)$ into the computation, it can be shown that that is not the case—the added error actually is $\mathrm{O}(h^3)$, and so that's the overall error.

For coding, it's helpful to split the RK2 method of Eq. (8) into three equations:

$$\begin{aligned} k_1 &= hf(x,t), \\ k_2 &= hf(x + \tfrac{1}{2}k_1, t + \tfrac{1}{2}h), \\ x(t+h) &= x(t) + k_2. \end{aligned}$$

With these equations at hand, code for implementing the RK2 method is easy to construct:

```
def RK2(f, a, b, N):
    '''Use 2nd-order Runge-Kutta method to integrate the function
    dx/dt = f(x,t) in the interval t = (a,b) using N steps.'''

    from numpy import linspace

    h = float((b-a)/N)            # time step size
    tpoints = linspace(a, b, N)   # time values
    xpoints = []
    x = a                         # initial value

    # RK2 method
    for t in tpoints:
        xpoints.append(x)
        k1 = h * f(x,t)
        k2 = h * f(x + 0.5*k1, t + 0.5*h)
        x = x + k2

    return tpoints, xpoints
```

While the 2^{nd}-order RK method works pretty well, one can extend the approach further to get higher-order methods that are more accurate. The most common version found in standard software library packages is the 4^{th}-order Runge-Kutta method [RK4, with $\mathrm{O}(h^5)$ error], one of the best-known computer algorithms of all. It is derived by performing Taylor expansions around several points and making linear combinations of them that eliminate terms of order h^3 and h^4. This process results in the following set of equations, which parallel those for RK2:

$$
\begin{aligned}
k_1 &= hf(x,t), \\
k_2 &= hf(x+\tfrac{1}{2}k_1, t+\tfrac{1}{2}h), \\
k_3 &= hf(x+\tfrac{1}{2}k_2, t+\tfrac{1}{2}h), \\
k_4 &= hf(x+k_3, t+h), \\
x(t+h) &= x(t) + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).
\end{aligned} \tag{9}
$$

These equations integrate the function by combining the derivatives evaluated at the initial point (t), at the endpoint ($t+h$), and twice at the midpoint ($t+\frac{1}{2}h$).

Several techniques even more accurate than RK4 exist for solving differential equations; for example, the ***predictor-corrector*** (or ***multistep***), and ***Bulirsch-Stoer*** methods. Since the 4^{th}-order RK method is generally quite effective, however, we will not explore those other methods.[2]

The `scipy` library has several functions for integrating ordinary differential equations, including one that can implement the RK4 method (as well as some of the more accurate approaches): `scipy.integrate.ode()`. A simpler integrator function is `scipy.integrate.odeint()`. See the `scipy` online reference pages for details on how to use these functions.

10.2 Second-Order Equations of One Variable

Given a method for solving first-order differential equations, extending it to second-order equations is made straightforward by using a simple "trick." The general form of equation to be solved is

$$
\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right). \tag{10}
$$

The "trick' is to define a new variable,

$$
z \equiv \frac{dx}{dt}, \tag{11}
$$

so that we can rewrite the original equation as

$$
\frac{dz}{dt} = f\left(x, z, t\right). \tag{12}
$$

By making this substitution, we have turned the original second-order equation into two first-order equations, (11) and (12), which can be solved by the methods of the previous section. This approach can be extended to even higher-order differential equations.

[2] For details on these methods, see *Numerical Recipes 3e*, by Press, Teukolsky, Vetterling, and Flannery.

Breakpoint 1: Use the trick described above to break up the pendulum equation, $\dfrac{d^2\theta}{dt^2} = -\dfrac{g}{l}\sin\theta$ into two first-order differential equations.

As an illustration of how to apply this trick, consider the familiar example of vertical motion of a projectile near the surface of the Earth. The second-order equation of motion is $d^2x/dt^2 = -g$, which can be written as two first-order equations $dx/dt = v$ and $dv/dt = -g$. Code to integrate these two equations simultaneously and compute the motion of the projectile is shown below (and discussed following the code block).

```
from numpy import arange, array

# Constants
a = 0.
b = 10.
N = 100
h = (b - a)/N
x0 = 0.
v0 = 20.

# Return the derivatives of x and v
def f(r, t):
    g = 9.8
    x = r[0]
    v = r[1]
    dx = v     # derivative of x
    dv = -g    # derivative of v
    return array([dx, dv], float)

# Implement RK4 integration
tpoints = arange(a, b, h)
xpoints = []
vpoints = []
r = array([x0, v0], float)

for t in tpoints:
    xpoints.append(r[0])
    vpoints.append(r[1])
    k1 = h * f(r, t)
    k2 = h * f(r + 0.5*k1, t + 0.5*h)
    k3 = h * f(r + 0.5*k2, t + 0.5*h)
    k4 = h * f(r + k3, t + h)
    r = r + (k1 + 2*k2 + 2*k3 + k4)/6

plot(tpoints,xpoints)
```

After declaring constants, the code defines a function `f(r,t)` to compute the derivatives indicated in the first-order equations; here, `r` is a two-element array representing the position variable `x` and the velocity variable `v`, and `dx` and `dv` represent the derivatives that the function returns. (This `f`

can serve as a model for any other function you might need to write.) The last bit of code uses the RK4 algorithm (with `r = [x, v]` as input) of Eqs. (9) to integrate the two first-order equations.

Simultaneous ordinary differential equations of the form $\frac{dx}{dt} = f_x(x, y, t)$, $\frac{dy}{dt} = f_y(x, y, t)$ can be solved in a similar way, by defining an appropriate derivative function `f(r,t)`.

10.3 Boundary Value Problems

The preceding examples represent the category of ***initial value*** (IV) ***problems***, in which the differential equations are integrated starting from some initial value. A different category involves applications in which we wish to solve a differential equation subject to specified values at *two* (or more) points; e.g., at two different times or at two different locations. These are called ***boundary value*** (BV) ***problems***. Because they involve matching multiple conditions, they generally are harder to solve than initial value problems.

10.3.1 The Shooting Method

A common technique for solving BV problems is known as the ***shooting method***, which uses a directed trial-and-error approach to find the initial condition that produces a solution that matches the specified boundary conditions. As an example, we might want to determine the proper speed with which to toss a ball upward so that it returns to its launch height (set as $y = 0$) at a specified later time (call it t_f). A simple guess of the speed is almost certain to lead to the ball's projected position being either too high or too low at the specified later time. How, then, should the guess be modified to arrive at a better estimate for the speed? Suppose we write the motion of the ball in the form $y = f(v)$, where y is its height at time t_f as a function of the launch speed v. Then, if we want to find the v such that $y = 0$ at t_f, we need to find the *root* of the motion function; i.e., we want to solve for the particular $v = v_*$ that satisfies $f(v_*) = 0$. There are many approaches to root finding, including the binary search method described briefly (in a different context) in Module 2, so this represents a viable approach to solving this motion problem. The solution involves two main steps:

1. Guess an initial speed v and use an integrator, e.g. RK4, to compute the corresponding position of the ball at time t_f. For a second-order differential equation, one needs to use the trick of section 10.2 to transform the equation into two first-order equations.

2. Use a root-finding method that repeatedly calls the integrator, with a shrinking range of values for the initial speed, to identify the one that produces motion matching the boundary conditions.

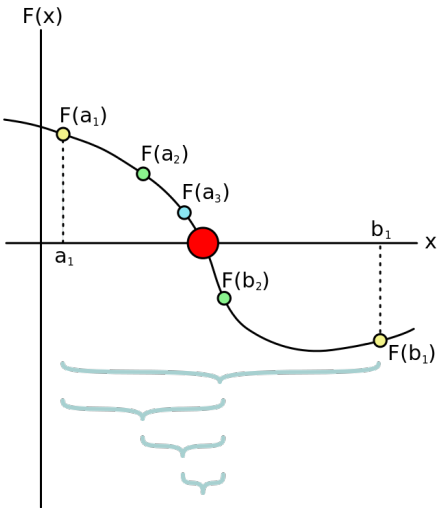


Figure 2: Bisection method of root-finding

Example #1

For the problem of the tossed ball discussed above, the differential equation of motion is $\frac{d^2y}{dt^2} = -g$, which can be written as the two first-order equations $z = \frac{dy}{dt}$, $\frac{dz}{dt} = -g$. Below is code that will solve this problem. After defining some constants, it creates a function to return the derivatives of y and z—those are passed into a specialized RK4 function to compute the final height (at time t_f) of the ball for a given initial speed. Subsequent code uses a binary search, a.k.a. ***bisection*** approach, which calls the RK4 function repeatedly with different initial speed values to compute the corresponding final heights of the ball, until it finds one that lies within a specified target range of zero height.

The bisection approach to finding a root of a function $F(x)$ starts with two "limiting" values of x, call them a_1 and b_1, that bound an interval known to contain the root. (See Figure 2.[3] The root is shown as a big red dot.) Such a choice of limiting values means that $F(x) < 0$ for one of them and $F(x) > 0$ for the other. The x value midway between the limiting values, $x_m = (x_1 + x_2)/2$, will become a new limiting value, replacing either x_1 or x_2—whichever produces an $F(x)$ value with the same sign as $F(x_m)$. [In the example shown in the figure, b_2 is the midway point between a_1 and b_1, and it replaces b_1 as a limiting point in the first iteration since $F(b_2)$ has the same sign as $F(b_1)$.] The process continues until the range containing the root is as small as desired. (The shrinking ranges are indicated by braces at the bottom of the figure.)

Breakpoint 2: What is the sequence of midpoint values x_m for the example shown in the figure (starting with b_2)?

[3]Image credit: "Bisection method" by Bisection_method.svg: Tokuchanderivative work: Tokuchan (talk)-Bisection_method.svg. Licensed under CC BY-SA 3.0 via Wikimedia Commons.

The following four code blocks together implement the shooting method for the tossed-ball problem.

```
# Initialization
from numpy import array, arange

g = 9.81           # Acceleration due to gravity
t0 = 0.0           # Initial time
tf = 10.0          # Final time
N = 1000           # Number of Runge-Kutta steps
h = (tf - t0)/N    # step size
target = 1.0E-10   # Target accuracy for binary search
```

Function to compute the derivatives $\frac{dy}{dt} = z$ and $\frac{dz}{dt} = -g$. The y and z values are input as a two-element array, `r`.

```
# Derivative functions for the two variables
def f(r):
    [y, z] = r     # vector of input variables
    dy = z         # derivative of y variable
    dz = -g        # derivative of z variable
    return array([dy,dz], float)
```

Next, Runge-Kutta 4$^{\text{th}}$-order integration is used to find the final height. The code simultaneously works on y and z, combined into a vector `r`. It integrates the motion starting from ground level (the `0.0` value for y in the declaration of `r`) using the speed `v` with which the function is called. Note that only y, i.e. `r[0]`, is returned, since it represents the final height, but both variables must be integrated because the position depends on the changing velocity, just as in the Euler integration of projectile motion in Module 3.

```
# Specialized RK4 function to calculate final height
def height(v):
    r = array([0.0, v], float)    # Initialize y and z
    for t in arange(t0, tf, h):   # The RK4 equations follow
        k1 = h * f(r)
        k2 = h * f(r + 0.5*k1)
        k3 = h * f(r + 0.5*k2)
        k4 = h * f(r + k3)
        r += (k1 + 2*k2 + 2*k3 + k4) / 6
    return r[0]
```

Below is the code to perform the binary search. It starts by setting limiting initial speed values, `v1` and `v2`, so that the corresponding final heights of the ball, `h1` and `h2`, obey `h1` < 0 and `h2` > 0. The particular speed we're looking for (call it $\mathtt{v}_*$) which will result in a final height $\mathtt{h}_* = 0$ therefore is guaranteed to lie between `v1` and `v2`. While the difference in heights, `abs(h2 - h1)`, is greater than the desired accuracy ("`target`"), the code executes the binary search in the `while` loop. In this search, the code identifies the midpoint (`vm`) of the current speed range and computes the corresponding final height of the ball (`hm`). If `hm` is positive (negative), the code replaces with

vm whichever of v1 or v2 also results in a positive (negative) height. In this way, the speed range surrounding v_* is narrowed down to the specified target accuracy. Once that's done, the midpoint of the range is identified as the approximation to v_*.

```
# Procedure to perform binary search
v1 = 0.01          # Lower limit of v range
v2 = 1000.0        # Upper limit of v range
h1 = height(v1)    # Final height corresponding to v1
h2 = height(v2)    # Final height corresponding to v2

while abs(h2-h1) > target:  # range containing v is too large
    vm = (v1 + v2) / 2      # midpoint of range v1:v2
    hm = height(vm)         # corresponding height

    if h1 * hm > 0:         # hm and latest h1 have same sign, so...
        v1 = vm             # vm replaces v1
        h1 = hm
    else:                   # hm and latest h1 have opposite signs, so...
        v2 = vm             # vm replaces v2
        h2 = hm

v = (v1 + v2) / 2           # return midpoint of range as v*
print("The required initial velocity is",v,"m/s")
```

Example #2

Another instance in which the shooting method can be used is in the solution of the (time-independent) Schrödinger equation,

$$-\frac{\hbar^2}{2m}\frac{d^2\Psi}{dx^2} + V(x)\Psi(x) = E\Psi(x). \tag{13}$$

Following the usual procedure, we convert this second-order equation to two first-order equations by introducing a new variable (ϕ) that obeys:

$$\frac{d\Psi}{dx} = \phi, \qquad \frac{d\phi}{dx} = \frac{2m}{\hbar^2}\left[V(x) - E\right]\Psi. \tag{14}$$

As a specific example, consider a particle in an infinite potential well, with potential $V(x)$ inside the well (in the range $0 < x < L$), and $V(x) = \infty$ elsewhere. We know that the wavefunction $\Psi(x)$ must vanish at $x = 0$ and $x = L$, where the potential becomes infinite. Thinking about the tossed-ball example, we might imagine that matching the boundary conditions would require us to specify the derivative of the wavefunction, $d\Psi/dx$ at $x = 0$ (analogous to the launch speed of the ball), but it turns out to be irrelevant. To understand why, imagine that we set $\Psi(0) = 0$ and assume some value for $d\Psi(0)/dx$ that leads not to $\Psi(L) = 0$, but to $\Psi(L) = \Delta$. Since the Schrödinger equation is linear, changing $d\Psi(0)/dx$ by some factor a will change $\Psi(L)$ to $a\Delta$. It won't be possible to force $\Psi(L) = 0$ without setting $d\Psi(0)/dx = 0$, which then would make $\Psi(x) = 0$ everywhere.

The problem here is that we can't match the boundary conditions for any old energy E: the Schrödinger equation will have solutions only for particular values of E—the energy eigenvalues. Thus, in this situation the shooting method is used to find those energies.

The code to solve this situation (see below) is similar in general form to that for Example #1, with some modifications. In the `f(r,x,e)` function, the two variables bundled into the `r` vector are "psi" (Ψ) and "phi" (ϕ). The derivatives of Ψ and ϕ used in the function correspond to the expressions in Eq. (14). In the `solve` function, $\Psi(0)$ is set to `0.0` and, as discussed above, the initial value for $\phi = d\Psi/dx$ can be chosen arbitrary (as long as it's not 0), so the code uses `1.0`. This function uses RK4 to solve for Ψ at the given energy; even though we ultimately care only about Ψ, both it and ϕ have to be integrated simultaneously, since the former function depends on the latter. The last section of code then uses the bisection approach to find the energy that makes $\Psi(L) = 0$. (Note that this isn't necessarily the best root-finding method in this situation. Since the initial limiting values of E used in the bisection method have to "surround" the true value, we have to have a sense of what that value is at the start. If we want to find the ground-state energy, a better root-finding approach would be the ***secant method***, for which the two initial values do not have to surround the true value. Thus, both could be chosen smaller than any ground state energy we might expect, and we wouldn't have to have a sense of that energy to begin with.)

To compare with the quantum exercise of Module 8, consider a well with potential $V(x) = ax/L$, where $a = 10$ eV and $L = 0.5$ nm. Here is a function that will find the ground-state energy:

```
def wellEnergy(e1, e2, a = 10., L = 5.0E-9):

    from numpy import array,arange

    # Constants
    m = 9.1094e-31     # mass of electron
    hbar = 1.0546e-34  # Planck's constant / 2*pi
    eV = 1.6022e-19    # 1 eV, in Joules (eV to Joules conversion)
    aeV = a * eV       # potential well constant
    N = 1000
    h = L/N

    # Potential function inside wel
    def V(x):
        return aeV*x/L

    # Derivatives
    def f(r, x, E):
        psi = r[0]
        phi = r[1]
        fpsi = phi
        fphi = (2*m / hbar**2) * (V(x) - E) * psi
        return array([fpsi,fphi],float)

    # Calculate the wavefunction for a particular energy
```

```
    def solve(E):
        psi = 0.0
        phi = 1.0
        r = array([psi,phi],float)

        for x in arange(0,L,h):
            k1 = h * f(r, x, E)
            k2 = h * f(r + 0.5*k1, x + 0.5*h, E)
            k3 = h * f(r + 0.5*k2, x + 0.5*h, E)
            k4 = h * f(r + k3, x+h, E)
            r += (k1 + 2*k2 + 2*k3 + k4)/6

            return r[0]

    # Main program to find the energy using the bisection method
    E1 = e1 * eV
    E2 = e2 * eV
    psi1 = solve(E1)
    psi2 = solve(E2)

    target = eV/100
    while abs(psi1 - psi2) > target:  # range containing E is too large
        Em = (E1 + E2) / 2    # midpoint of range E1:E2
        psim = solve(Em)      # corresponding psi function

        if psi1 * psim > 0:   # Em and latest E1 have same sign, so...
            E1 = Em           # Em replaces E1
            psi1 = psim
        else:                 # Em and latest E1 have opposite signs, so...
            E2 = Em           # Em replaces E2
            psi2 = psim

    # Once the range containing v is narrow enough,
    # return the midpoint of that range as the desired v
    E = (E1 + E2) / 2

    print("The energy eigenvalue is", E/eV, "eV")
```

Recap

- Euler's method for integrating ordinary differential equations is one of a series of increasingly accurate Runge-Kutta methods. RK4 (with $O(h^5)$ error) is commonly used.
- A second-order equation can be converted to two first-order equations simply by introducing a new variable whose first derivative is the second derivative of the original function.
- One way to solve boundary value problems is using the shooting method, a directed trial-and-error approach based on root-finding.

Exercises

Exercise #1

(a) Port the code for the function `RK2` into an IPython notebook and try it out on the function $f(x,t) = -x^3 + \sin(t)$ over the range $t = (0, 10)$ using 100 timesteps. (b) Replace the appropriate lines in `RK2` with the five lines that implement the 4^{th}-order RK method, to make a new `RK4` function, and use it on the test function of part (a). Plot both sets of results on one graph, and their difference on a separate graph. You might also want to compare with the output of `scipy.integrate.odeint()`.

Exercise #2

(a) Use RK4 integration to compute and plot the 1-D motion of an object moving without acceleration. Apply it to an object started with a speed of 20.0 (m/s).

(b) Repeat for vertical motion with the standard acceleration due to gravity. [You will need to use the "trick" discussed in the Module for converting a 2nd-order equation to two 1st-order ones. The two 1st-order equations can be integrated by your RK4 code at the same time, by incorporating them and their derivatives into two-element arrays or lists, as in Example #2 in section 10.3.1 of the Module.] Compute the motion over 5 (seconds).

Exercise #3

The equation of motion of a simple pendulum of length l with angular position θ is $\dfrac{d^2\theta}{dt^2} = -\dfrac{g}{l}\sin\theta$. This equation cannot be solved analytically, so we typically simplify (*linearize*) it using the small-angle approximation, according to which $\sin\theta \simeq \theta$. However, the exact equation *can* be solved numerically. Do that with your RK4 code by using the trick described earlier, for $l = 0.1$ m and 200 time steps. Also solve the linearized equation. Plot both results on the same graph. Briefly describe and try to explain the differences.

Breakpoint Answers

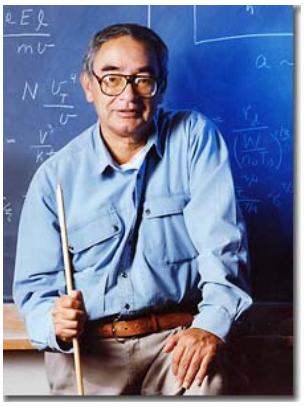
Breakpoint 1

We define a new variable $\phi \equiv \dfrac{d\theta}{dt}$, and then the pendulum equation states $\dfrac{d\phi}{dt} = -\dfrac{g}{l}\sin\theta$. These are the desired two first-order equations.

Breakpoint 2

The sequence is b_2 (halfway between a_1 and b_1), a_2 (halfway between a_1 and b_2), and a_3 (halfway between a_2 and b_2).

Scientist Profile



Dr. Fred Begay was born on the Ute Mountain Indian Reservation in Colorado in 1932. At the age of 10 he was sent to a government-run school to learn farming; students there were forbidden from practicing the Navajo language and culture, and were forced to speak English and attend a Christian church. At age 18 (in 1951) he left to join the Air Force and fight in the Korean War. Returning to the U.S. in 1955, he enrolled at the University of New Mexico, where his decision to study physics was "just an accident." He went on to earn a PhD in nuclear physics from the university in 1972, and subsequently joined the research staff of Los Alamos National Laboratory. His research has focused on controlled thermonuclear fusion in plasmas, and on the use of lasers for heating such plasmas. Begay credits his Navajo culture for allowing him to learn physics despite a very unusual school experience: "I think the key point is that I learned to think abstractly and develop reasoning skills when I was growing up, learning about lasers and radiation in the Navajo language" he says, "That's all embedded in our religion." Begay has participated in many outreach and education efforts to support the Native American community. (Profile adapted from http://www.physicscentral.com/explore/people/begay.cfm.)