ASTR 119: Session 11
Ordinary Differential Equations

Outline

1) Homework due 11/12 at 8:00am
2) Visualization of the Day
3) Ordinary Differential Equations
4) Save your work to GitHub

Homework, due Nov 12, 8:00am

1) Repeat the exercise from the 11/6 session, but use the Cash-Karp Runge-Kutta method with adaptive stepwise control

2) Evolve the system of equations:

$$\frac{dy}{dx} = z \qquad \frac{dz}{dx} = -y$$

Use the initial conditions y(x=0) = 0 and dydx(x=0) = 1, and evolve over the range [0, 2*pi].

3) Plot the analytical solutions for y(x) and dy/dx(x) over the specified range, and the numerical solution.

4) Plot the absolute error for the numerical solutions of y(x) and dy/dx(x) over the specified range.

5) Call the repository “astr-119-hw-6” and the notebook “hw-6.ipynb”.

Computing/Programming Check-In

1) Create a jupyter notebook, and name it astr-119-code-check-in.ipynb.

2) Add a markdown line that describes each cell in the notebook. Each of the following instructions should be its own cell, in order.

3) Import numpy and matplotlib.pyplot as usual.

4) Declare integer i, set equal to zero. Declare a float x, set equal to 119.

5) Use a for loop, iterate i from 0 to 119 inclusive. For each even value (including 0) of i, add 3 to x. For each odd value of i, subtract 5 from x.

6) Print the final value of x in scientific notation using 2 decimal places.

You will have ~ 5 minutes.

TAs will be sending sign-up information, starting in section next week.

ORDINARY DIFFERENTIAL EQUATIONS

What is an ODE? **Consider a function f(t) that is depends only on a single variable t. A differential equation is an equality involving one or more derivatives of the function along with any combination of the function itself and the independent variable. This equality holds for any value of the independent variable. Here are some examples:**

$$\frac{df}{dt} + f^3 = ft^2$$

$$\frac{d^2f}{dt^2} + t^3\frac{df}{dt} = ft^2$$

To solve the ODE: find f(t) for all t that satisfies the equality.

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

You know the solution:

$$f(t) = \frac{1}{3}t^3 + C$$

ODEs: BOUNDARY CONDITIONS

Consider the following ODE:

$$\frac{df}{dt} = t^2$$

You know the solution:

$$f(t) = \frac{1}{3}t^3 + C$$

To determine C, we must have a boundary condition that specifies $f(t_0)$ at some time t_0. Since there are infinitely many solutions involving infinitely many choices of C, when we numerically solve ODEs we must always adopt some boundary conditions as part of the solution process.

ORDINARY DIFFERENTIAL EQUATIONS

In physics, we often describe the state of a system as a function of some unknown variables. For instance, the vertical motion of a projectile under the influence of gravity can be written:

$$y(t) = y_0 + v_0 t + \frac{1}{2} g t^2$$

However, the same physical system can be described in terms of differential equations:

$$\dot{y} = v(t)$$

$$\dot{v} = g$$

with the boundary conditions:

$$v(t = 0) = v_0$$

$$y(t = 0) = y_0$$

Finite Difference

Consider the differential equation:

$$f' \equiv \frac{df}{dx} = g(x, f), \quad f(x_0) = f_0$$

We want to solve for values $f(x_i)$ on a grid x_i, i=[1,n].

We will notate $f(x_i)$ as f_i and $f(x_{i+1})$ as f_{i+1}. Symbolically, we then want

$$f_i, x_i, g(x_i, f_i) \rightarrow f_{i+1}$$

How NOT to Evolve an ODE

Consider the discretized equation:

$$f_{i+1} = f_i + hg(x_i, f_i) + \frac{h^2}{2!} f''(\xi)$$

While this initially seems sensible, note that the error term for each step is quadratic in *h*. After many steps, the overall error will only improve as *h*, and that is not good! We need to search out better methods.

SECOND-ORDER RUNGE-KUTTA METHOD

We can produce a more accurate method by taking a half-step, $f_{i+1/2}$:

$$f_{i+1/2} = f_i + \frac{h}{2} f_i' + O(h^2 f'')$$

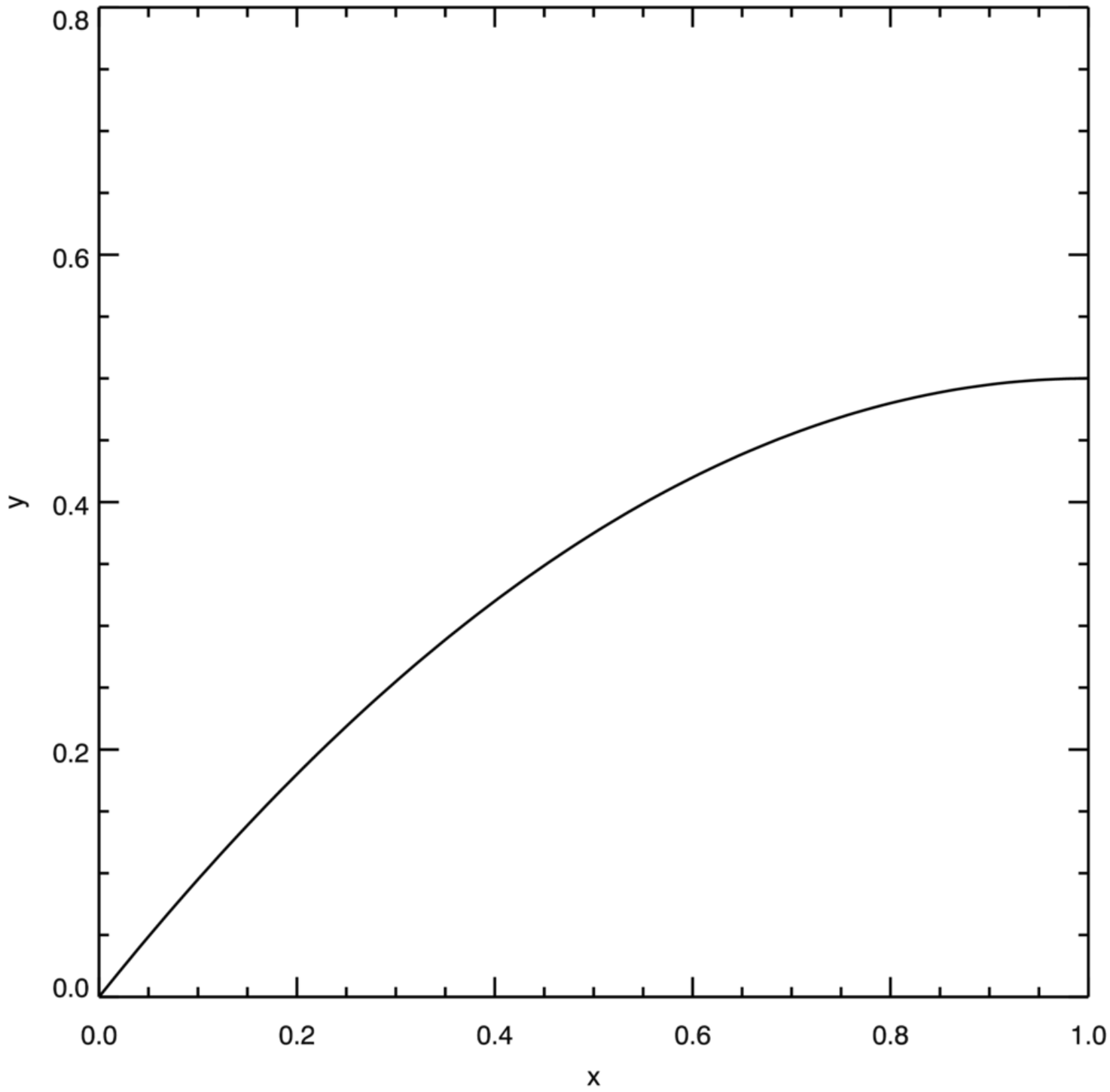
All terms involving $f_{i+1/2}$ in our approximation to f_i are multiplied by h and we preserve an error $O(h^3)$. Here are the equations that we need to evaluate:

$$f_{i+1/2} = f_i + \frac{h}{2} g(x_i, f_i)$$

$$f_{i+1} = f_i + h g(x_{i+1/2}, f_{i+1/2}) + O(h^3)$$

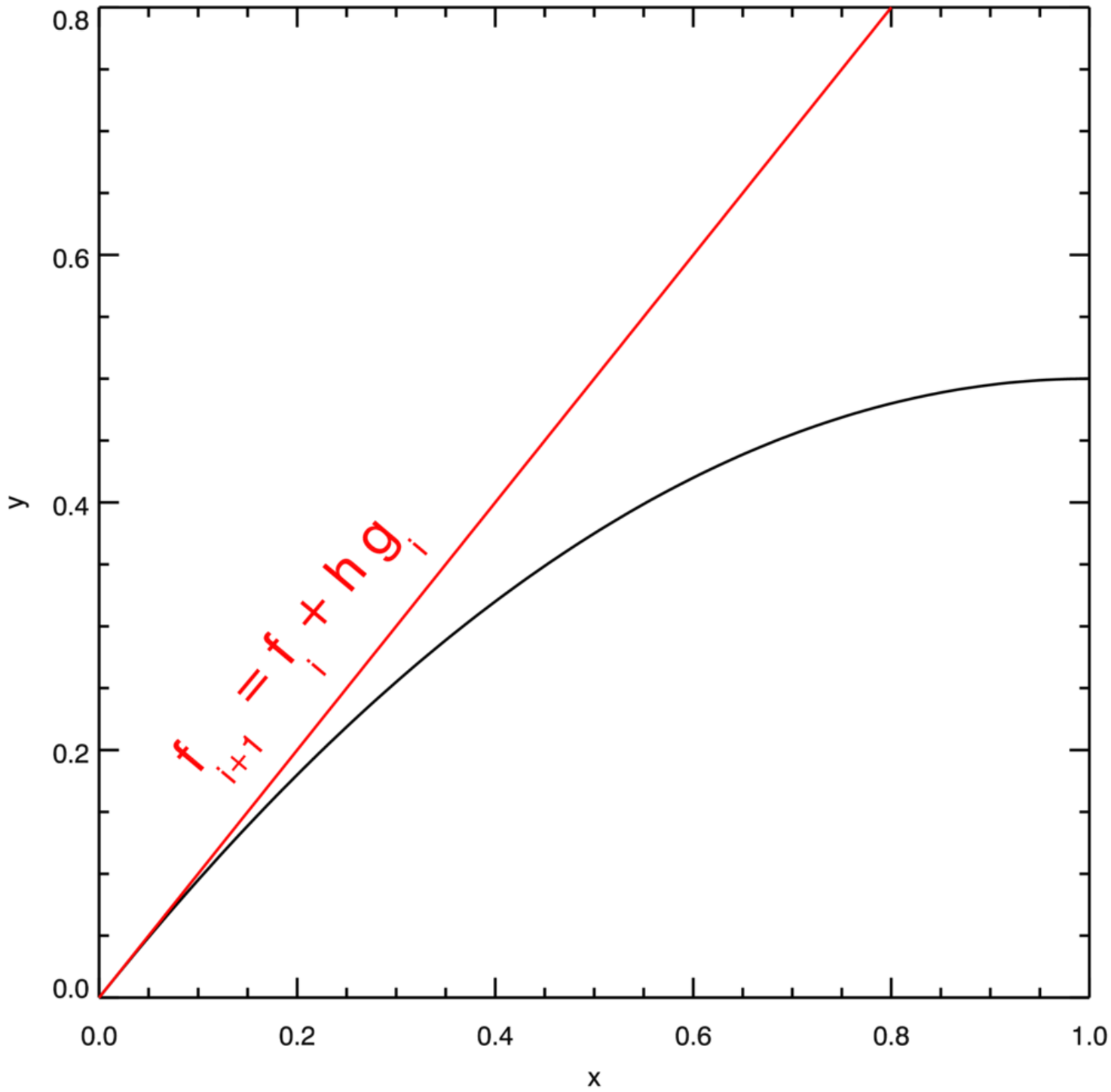
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



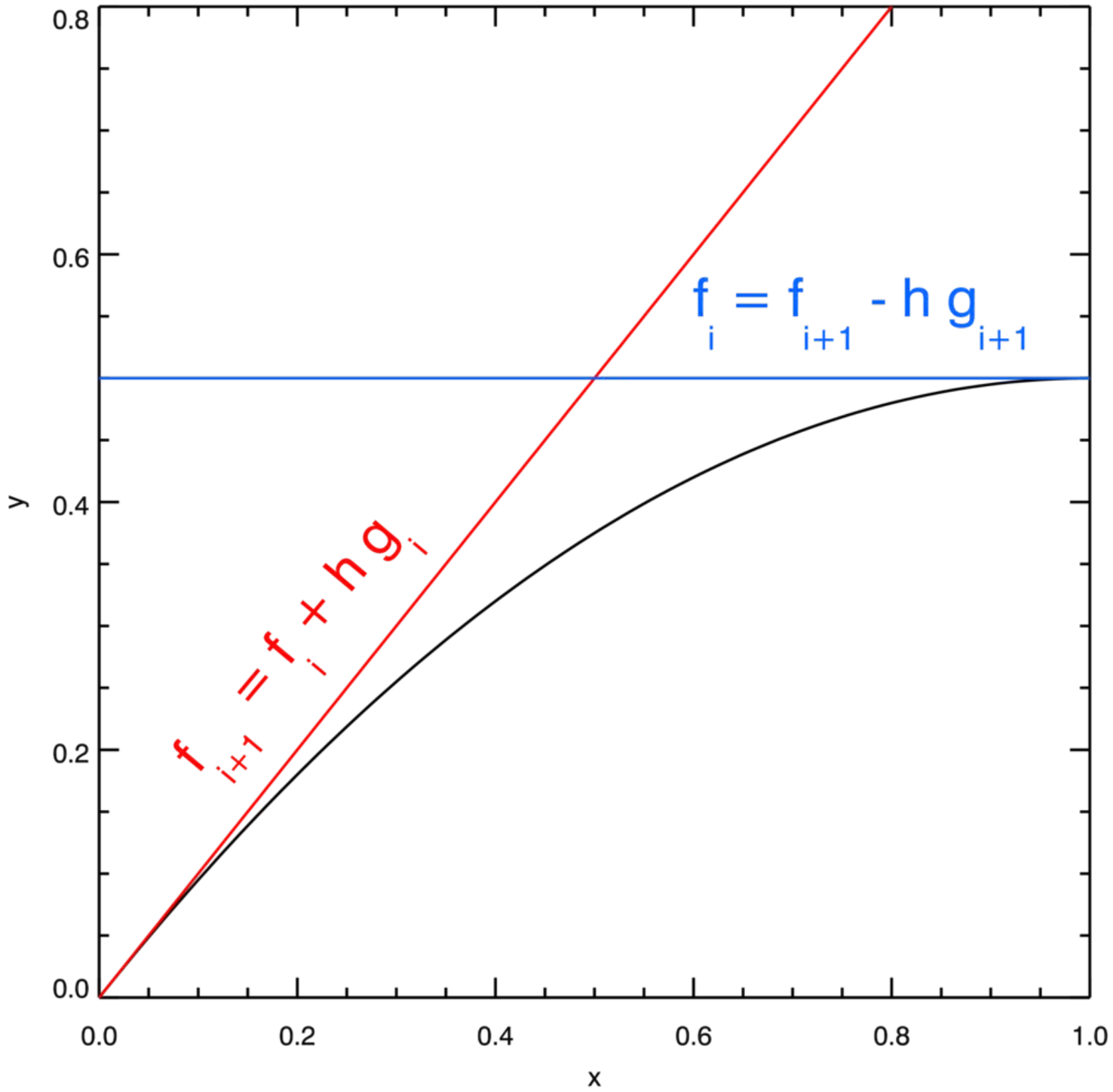
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



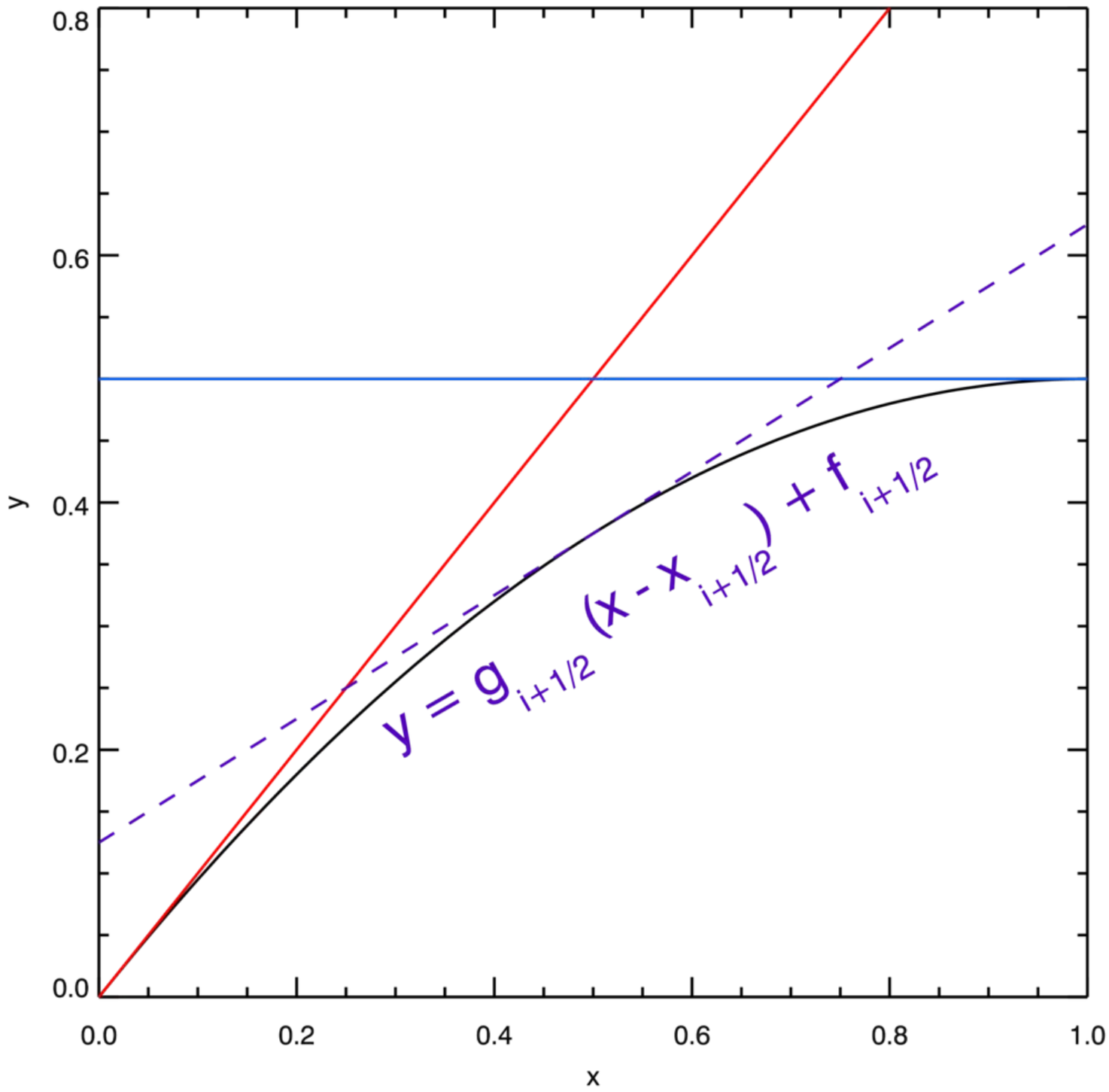
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



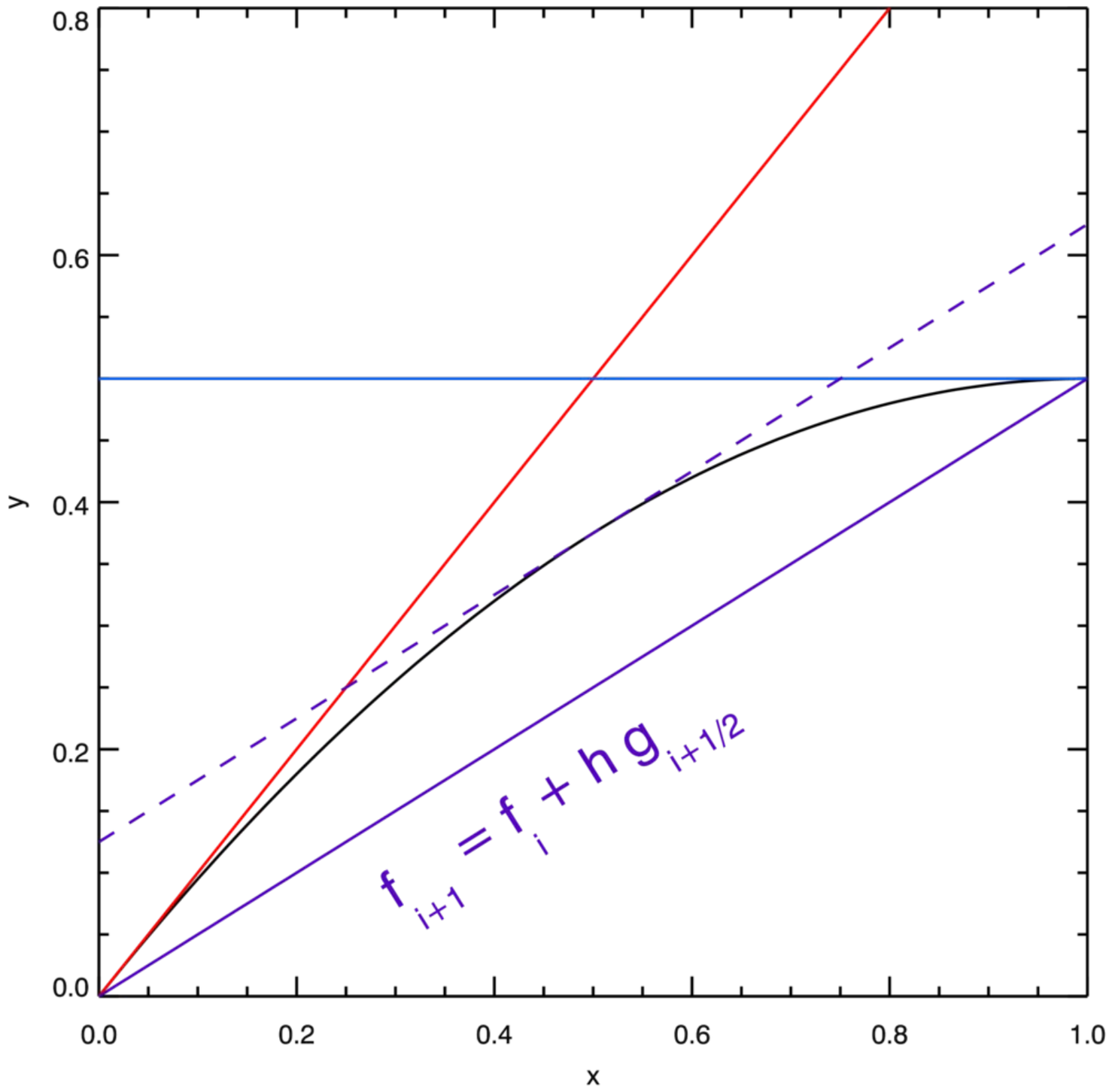
This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



This is called the second-order Runge-Kutta method or the midpoint method.

SECOND-ORDER RUNGE-KUTTA METHOD



This is called the second-order Runge-Kutta method or the midpoint method.

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

$$k_4 = hg\left(x_{i+1}, f_i + k_3\right)$$

FOURTH-ORDER RUNGE-KUTTA METHOD

You can imagine that this approach can be extended to even higher orders. Without showing the derivation, here is the fourth-order Runge-Kutta method:

$$k_1 = hg(x_i, f_i)$$

$$k_2 = hg\left(x_{i+1/2}, f_i + \frac{k_1}{2}\right)$$

$$k_3 = hg\left(x_{i+1/2}, f_i + \frac{k_2}{2}\right)$$

$$k_4 = hg\left(x_{i+1}, f_i + k_3\right)$$

$$f_{i+1} = f_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

Runge-Kutta Implementation

Create a notebook to perform Runge-Kutta integration.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Runge-Kutta Implementation

Define a function to integrate

In [11]:
```
1  def dfdx(x,f):
2      return x**2 + x
```

Define its integral

In [12]:
```
1  def f_int(x,C):
2      return (x**3)/3. + 0.5*x**2 + C
```

Runge-Kutta Implementation

Define the 2nd order RK method

In [20]:

```
1  def rk2_core(x_i,f_i,h,g):
2
3      #advance f by a step h
4
5      #half step
6      x_ipoh = x_i + 0.5*h
7      f_ipoh = f_i + 0.5*h*g(x_i,f_i)
8
9      #full step
10     f_ipo = f_i + h*g(x_ipoh, f_ipoh)
11
12     return f_ipo
```

Runge-Kutta Implementation

Define a wrapper routine for RK2

In [22]:

```
def rk2(dfdx,a,b,f_a,N):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #f_a is the boundary condition at a
    #N is the number of steps

    #define our steps
    x = np.linspace(a,b,N)

    #a single step size
    h = x[1]-x[0]

    #an array to hold f
    f = np.zeros(N,dtype=float)

    f[0] = f_a #value of f at a

    #evolve f along x
    for i in range(1,N):
        f[i] = rk2_core(x[i-1],f[i-1],h,dfdx)

    return x,f
```

Runge-Kutta Implementation

Define the 4th order RK method

In [21]:

```
def rk4_core(x_i,f_i,h,g):

    #define x at 1/2 step
    x_ipoh = x_i + 0.5*h

    #define x at 1 step
    x_ipo = x_i + h

    #advance f by a step h

    k_1 = h*g(x_i,f_i)
    k_2 = h*g(x_ipoh, f_i + 0.5*k_1)
    k_3 = h*g(x_ipoh, f_i + 0.5*k_2)
    k_4 = h*g(x_ipo,  f_i + k_3)

    f_ipo = f_i + (k_1 + 2*k_2 + 2*k_3 + k_4)/6.

    return f_ipo
```

Runge-Kutta Implementation

Define a wrapper for RK4

In [23]:

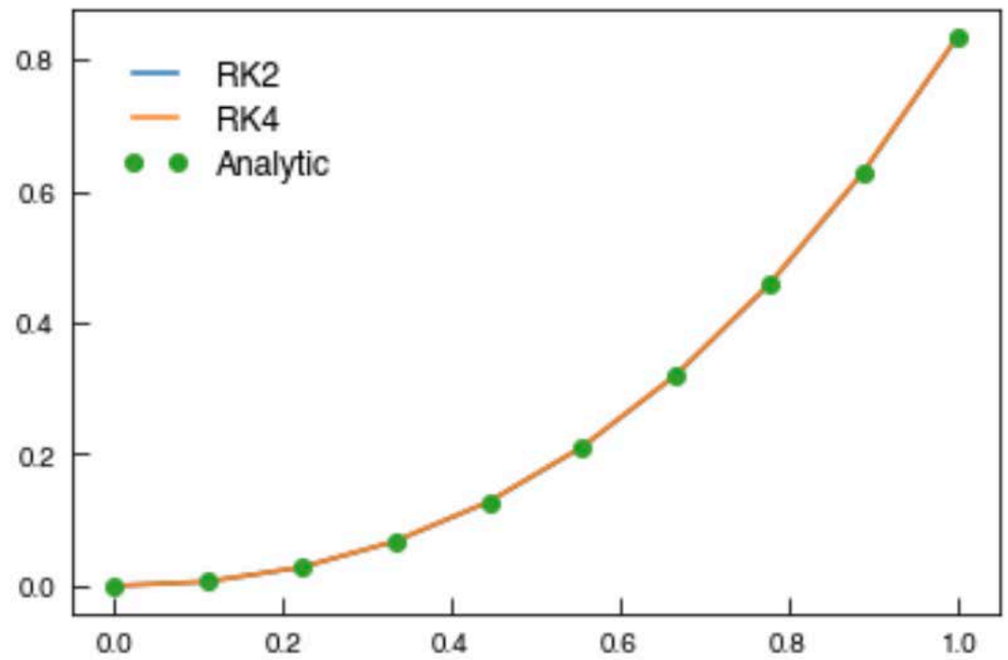
```
def rk4(dfdx,a,b,f_a,N):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #f_a is the boundary condition at a
    #N is the number of steps

    #define our steps
    x = np.linspace(a,b,N)

    #a single step size
    h = x[1]-x[0]

    #an array to hold f
    f = np.zeros(N,dtype=float)

    f[0] = f_a #value of f at a

    #evolve f along x
    for i in range(1,N):
        f[i] = rk4_core(x[i-1],f[i-1],h,dfdx)

    return x,f
```

Runge-Kutta Implementation

Perform the integration

In [10]:
```
a = 0.0
b = 1.0
f_a = 0.0
N = 10
x_2, f_2 = rk2(dfdx,a,b,f_a,N)
x_4, f_4 = rk4(dfdx,a,b,f_a,N)
x = x_2.copy()
plt.plot(x_2,f_2,label='RK2')
plt.plot(x_4,f_4,label='RK4')
plt.plot(x,f_int(x,f_a),'o',label='Analytic')
plt.legend(frameon=False)
```

Out[10]: <matplotlib.legend.Legend at 0x10f9e6ac8>



A Simple Coupled ODE

Here is a simple second-order ODE that may be familiar:

$$\frac{d^2y}{dx^2} = -y$$

A Simple Coupled ODE

Here is a simple second-order ODE that may be familiar:

$$\frac{d^2y}{dx^2} = -y$$

With the boundary conditions of y(0) = 0 and dy/dx(x0) = 1, the solution is sin(x).

A Simple Coupled ODE

Here is a simple second-order ODE that may be familiar:

$$\frac{d^2y}{dx^2} = -y$$

With the boundary conditions of y(0) = 0 and dy/dx(x0) = 1, the solution is sin(x).

We can re-write this equation as a set of coupled ODEs:

$$\frac{dy}{dx} = z$$

$$\frac{dz}{dx} = -y$$

With the boundary conditions y(0) = 0, z(0) = 1, we can solve this numerically adapting techniques we have already learned.

Coupled ODE Solver

jupyter runge_kutta_mv Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Help

Trusted | Python 3

Run Code

Create a notebook to perform Runge-Kutta integration for multiple coupled variables.

In [1]:

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import numpy as np
```

Coupled ODE Solver

Define our coupled derivatives to integrate

In [2]:

```
def dydx(x,y):

    #set the derivatives

    # our equation is d^2y/dx^2 = -y

    #so we can write
    #dydx = z
    #dzdx = -y

    #we will set y = y[0]
    #we will set z = y[1]

    #declare an array
    y_derivs = np.zeros(2)

    #set dydx = z
    y_derivs[0] =    y[1]

    #set dzdx = -y
    y_derivs[1] = -1*y[0]

    #here we have to return an array
    return y_derivs
```

Coupled ODE Solver

Define the 4th order RK method

In [3]:

```
def rk4_mv_core(dydx,xi,yi,nv,h):

    #declare k? arrays
    k1 = np.zeros(nv)
    k2 = np.zeros(nv)
    k3 = np.zeros(nv)
    k4 = np.zeros(nv)

    #define x at 1/2 step
    x_ipoh = xi + 0.5*h

    #define x at 1 step
    x_ipo = xi + h

    #declare a temp y array
    y_temp = np.zeros(nv)

    #get k1 values
    y_derivs = dydx(xi,yi)
    k1[:] = h*y_derivs[:]

    #get k2 values
    y_temp[:] = yi[:] + 0.5*k1[:]
    y_derivs = dydx(x_ipoh,y_temp)
    k2[:] = h*y_derivs[:]

```

Coupled ODE Solver

```
    #get k2 values
    y_temp[:] = yi[:] + 0.5*k1[:]
    y_derivs = dydx(x_ipoh,y_temp)
    k2[:] = h*y_derivs[:]

    #get k3 values
    y_temp[:] = yi[:] + 0.5*k2[:]
    y_derivs = dydx(x_ipoh,y_temp)
    k3[:] = h*y_derivs[:]

    #get k4 values
    y_temp[:] = yi[:] + k3[:]
    y_derivs = dydx(x_ipo,y_temp)
    k4[:] = h*y_derivs[:]


    #advance y by a step h
    yipo = yi + (k1 + 2*k2 + 2*k3 + k4)/6.

    return yipo
```

(Cell continued)

Coupled ODE Solver

Define an adaptive step size driver for RK4

In [4]:

```
def rk4_mv_ad(dydx,x_i,y_i,nv,h,tol):

    #define safety scale
    SAFETY     = 0.9
    H_NEW_FAC = 2.0

    #set a maximum number of iterations
    imax = 10000

    #set an iteration variable
    i = 0

    #create an error
    Delta = np.full(nv,2*tol)

    #remember the step
    h_step = h

    #adjust step
    while(Delta.max()/tol > 1.0):

```

Coupled ODE Solver

```
    #adjust step
    while(Delta.max()/tol > 1.0):

        #estimate our error by taking one step of size h
        #vs. two steps of size h/2
        y_2  = rk4_mv_core(dydx,x_i,y_i,nv,h_step)
        y_1  = rk4_mv_core(dydx,x_i,y_i,nv,0.5*h_step)
        y_11 = rk4_mv_core(dydx,x_i+0.5*h_step,y_1,nv,0.5*h_step)

        #compute an error
        Delta = np.fabs(y_2 - y_11)

        #if the error is too large, take a smaller step
        if(Delta.max()/tol > 1.0):

            #our error is too large, decrease the step
            h_step *= SAFETY * (Delta.max()/tol)**(-0.25)


        #check iteration
        if(i>=imax):
            print("Too many iterations in rk4_mv_ad()")
            raise StopIteration("Ending after i = ",i)

        #iterate
        i+=1


    #next time, try to take a bigger step
    h_new = np.fmin(h_step * (Delta.max()/tol)**(-0.9), h_step*H_NEW_FAC)
```

(Cell continued)

Coupled ODE Solver

(Cell continued)

```
44
45
46      #next time, try to take a bigger step
47      h_new = np.fmin(h_step * (Delta.max()/tol)**(-0.9), h_step*H_NEW_FAC)
48
49      #return the answer, a new step, and the step we actually took
50      return y_2, h_new, h_step
```

Coupled ODE Solver

Define a wrapper for RK4

In [5]:

```
def rk4_mv(dfdx,a,b,y_a,tol):

    #dfdx is the derivative wrt x
    #a is the lower bound
    #b is the upper bound
    #y_a are the boundary conditions
    #tol is the tolerance for integrating y

    #define our starting step
    xi = a
    yi = y_a.copy()

    #an initial step size == make very small!
    h = 1.0e-4 * (b-a)

    #set a maximum number of iterations
    imax = 10000

    #set an iteration variable
    i = 0

    #set the number of coupled odes to the
    #size of y_a
    nv = len(y_a)

```

Coupled ODE Solver

(Cell continued)

```
    #set the number of coupled odes to the
    #size of y_a
    nv = len(y_a)


    #set the initial conditions
    x = np.full(1,a)
    y = np.full((1,nv),y_a)

    #set a flag
    flag = 1

    #loop until we reach the right side
    while(flag):

        #calculate y_i+1
        yi_new, h_new, h_step = rk4_mv_ad(dydx,xi,yi,nv,h,tol)

        #update the step
        h = h_new

        #prevent an overshoot
        if(xi+h_step>b):

            #take a smaller step
            h = b-xi

            #recalculate y_i+1
            yi_new, h_new, h_step = rk4_mv_ad(dydx,xi,yi,nv,h,tol)

            #break
            flag = 0

```

Coupled ODE Solver

```
51
52              #break
53              flag = 0
54
55
56          #update values
57          xi += h_step
58          yi[:]  = yi_new[:]
59
60          #add the step to the arrays
61          x = np.append(x,xi)
62          y_new = np.zeros((len(x),nv))
63          y_new[0:len(x)-1,:] = y
64          y_new[-1,:] = yi[:]
65          del y
66          y = y_new
67
68          #prevent too many iterations
69          if(i>=imax):
70
71              print("Maximum iterations reached.")
72              raise StopIteration("Iteration number = ",i)
73
74
75          #iterate
76          i += 1
77
78          #output some information
79          s = "i = %3d\tx = %9.8f\th = %9.8f\tb=%9.8f" % (i,xi, h_step, b)
80          print(s)
81
82          #break if new xi is == b
83          if(xi==b):
84              flag = 0
85
86      #return the answer
87      return x,y
```

(Cell continued)

Coupled ODE Solver

Perform the integration

In [6]:

```
a = 0.0
b = 2.0 * np.pi

y_0 = np.zeros(2)
y_0[0] = 0.0
y_0[1] = 1.0
nv = 2

tolerance = 1.0e-6

#perform the integration
x,y = rk4_mv(dydx,a,b,y_0,tolerance)
```

Coupled ODE Solver

```
i =   1 x = 0.00062832  h = 0.00062832  b=6.28318531
i =   2 x = 0.00188496  h = 0.00125664  b=6.28318531
i =   3 x = 0.00439823  h = 0.00251327  b=6.28318531
i =   4 x = 0.00942478  h = 0.00502655  b=6.28318531
i =   5 x = 0.01947787  h = 0.01005310  b=6.28318531
i =   6 x = 0.03958407  h = 0.02010619  b=6.28318531
i =   7 x = 0.07979645  h = 0.04021239  b=6.28318531
i =   8 x = 0.16022123  h = 0.08042477  b=6.28318531
i =   9 x = 0.32107077  h = 0.16084954  b=6.28318531
i =  10 x = 0.46816761  h = 0.14709684  b=6.28318531
i =  11 x = 0.60535502  h = 0.13718741  b=6.28318531
i =  12 x = 0.74522296  h = 0.13986794  b=6.28318531
i =  13 x = 0.88873209  h = 0.14350913  b=6.28318531
i =  14 x = 1.02700188  h = 0.13826979  b=6.28318531
i =  15 x = 1.16350434  h = 0.13650247  b=6.28318531
i =  16 x = 1.29828215  h = 0.13477781  b=6.28318531
i =  17 x = 1.43207856  h = 0.13379641  b=6.28318531
i =  18 x = 1.56536018  h = 0.13328163  b=6.28318531
```

(snip)

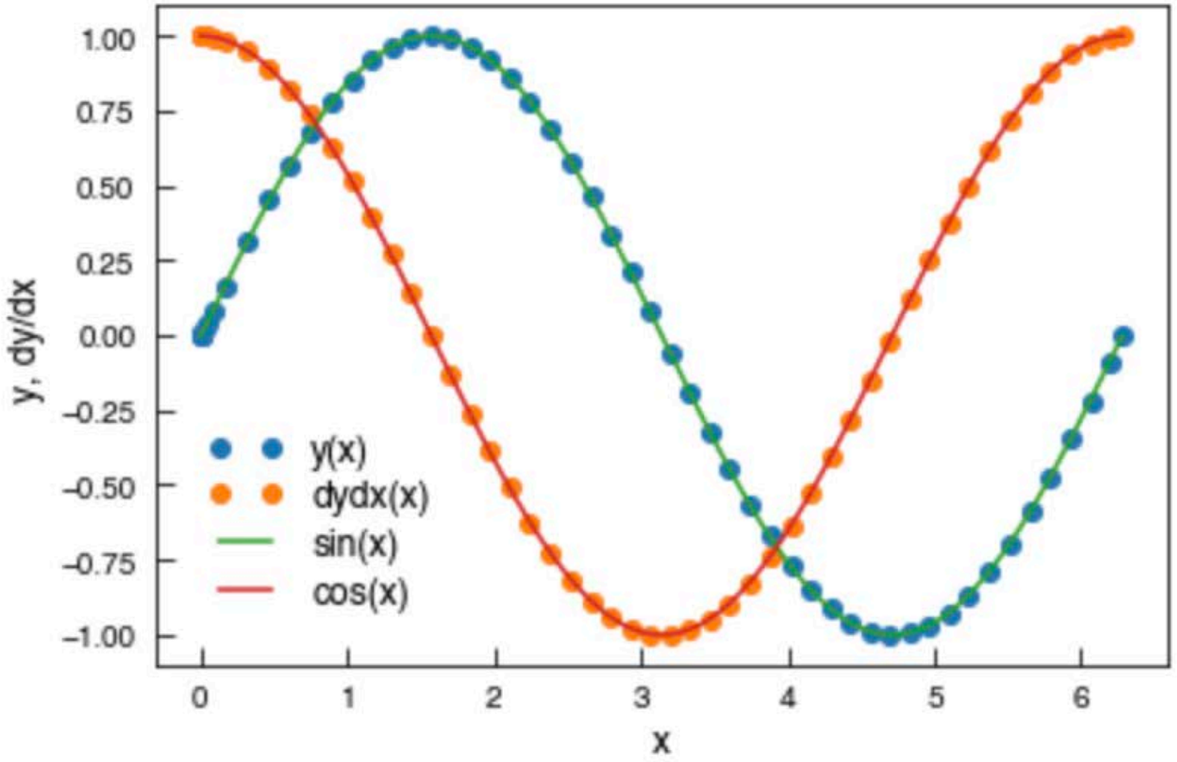
```
i =  39 x = 4.43060314  h = 0.13487723  b=6.28318531
i =  40 x = 4.56445058  h = 0.13384745  b=6.28318531
i =  41 x = 4.69775227  h = 0.13330169  b=6.28318531
i =  42 x = 4.83099353  h = 0.13324126  b=6.28318531
i =  43 x = 4.96464709  h = 0.13365356  b=6.28318531
i =  44 x = 5.09921335  h = 0.13456626  b=6.28318531
i =  45 x = 5.23525228  h = 0.13603892  b=6.28318531
i =  46 x = 5.37343256  h = 0.13818029  b=6.28318531
i =  47 x = 5.51460989  h = 0.14117733  b=6.28318531
i =  48 x = 5.65594750  h = 0.14133761  b=6.28318531
i =  49 x = 5.79342190  h = 0.13747440  b=6.28318531
i =  50 x = 5.92917073  h = 0.13574883  b=6.28318531
i =  51 x = 6.06349751  h = 0.13432679  b=6.28318531
i =  52 x = 6.19703289  h = 0.13353538  b=6.28318531
i =  53 x = 6.28318531  h = 0.08615241  b=6.28318531
```

Coupled ODE Solver

Plot the result

In [7]:
```
1  plt.plot(x,y[:,0],'o',label='y(x)')
2  plt.plot(x,y[:,1],'o',label='dydx(x)')
3  xx = np.linspace(0,2.0*np.pi,1000)
4  plt.plot(xx,np.sin(xx),label='sin(x)')
5  plt.plot(xx,np.cos(xx),label='cos(x)')
6  plt.xlabel('x')
7  plt.ylabel('y, dy/dx')
8  plt.legend(frameon=False)
```

Out[7]: <matplotlib.legend.Legend at 0x10f7a4198>



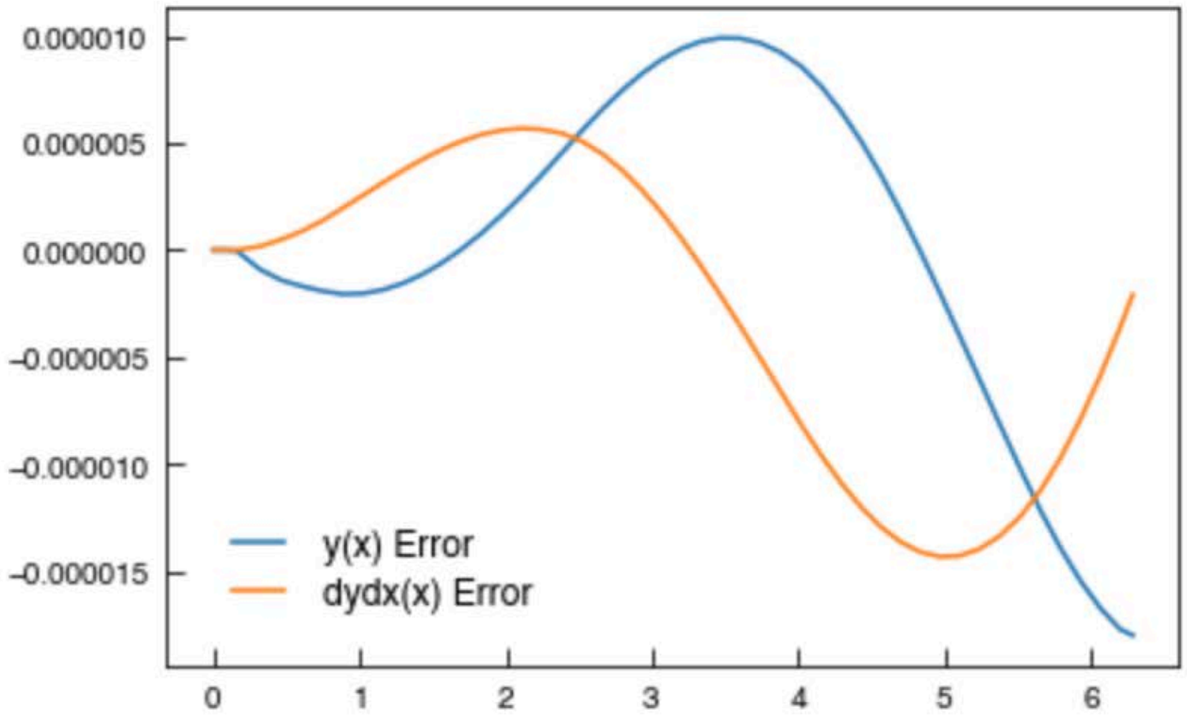
Coupled ODE Solver

Plot the error

Notice that the errors will actually exceed our "tolerance".

In [8]:
```
sine = np.sin(x)
cosine = np.cos(x)

y_error = (y[:,0]-sine)
dydx_error = (y[:,1]-cosine)

plt.plot(x, y_error, label="y(x) Error")
plt.plot(x, dydx_error, label="dydx(x) Error")
plt.legend(frameon=False)
```

Out[8]: <matplotlib.legend.Legend at 0x10f934fd0>



ADAPTIVE STEP-SIZE CONTROL

The problem with using this approach as presented is that monitoring the error $\Delta = y_2 - y_{1+1}$ requires performing two solutions simultaneously. What's nice for the Runge-Kutta method is that we can choose carefully crafted fourth and fifth order Runge-Kutta methods whose difference gives us an error estimate.

ADAPTIVE STEP-SIZE CONTROL

Consider the general form of the fifth-order Runge-Kutta method:

$$k_1 = hf(x_n, y_n) \qquad k_2 = hf(x_n + c_2 h, y_n + a_{21} k_1)$$

$$k_3 = hf(x_n + c_3 h, y_n + a_{31} k_1 + a_{32} k_2)$$

$$k_4 = hf(x_n + c_4 h, y_n + a_{41} k_1 + a_{42} k_2 + a_{43} k_3)$$

$$k_5 = hf(x_n + c_5 h, y_n + a_{51} k_1 + a_{52} k_2 + a_{53} k_3 + a_{54} k_4)$$

$$k_6 = hf(x_n + c_6 h, y_n + a_{61} k_1 + a_{62} k_2 + a_{63} k_3 + a_{64} k_4 + a_{65} k_5)$$

$$y_{n+1} = y_n + b_1 k_1 + b_2 k_2 + b_3 k_3 + b_4 k_4 + b_5 k_5 + b_6 k_6 + O(h^6)$$

$$y^\star_{n+1} = y_n + b^\star_1 k_1 + b^\star_2 k_2 + b^\star_3 k_3 + b^\star_4 k_4 + b^\star_5 k_5 + b^\star_6 k_6 + O(h^5)$$

$$\Delta \equiv y_{n+1} - y^\star_{n+1} = \sum_{i=1}^{6} (b_i - b^\star_i) k_i$$

ADAPTIVE STEP-SIZE CONTROL

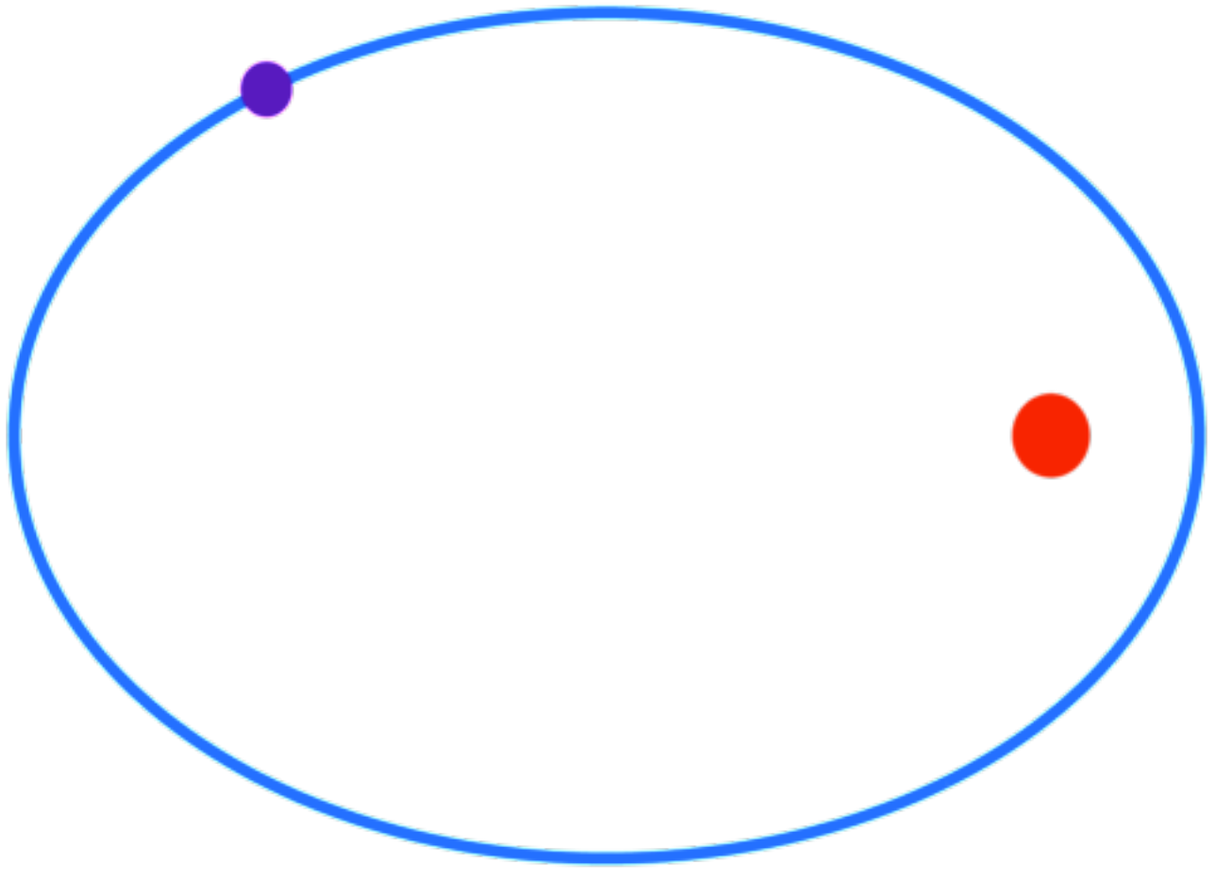
The key is to cleverly pick the coefficients c, a, and b:

i	c_i	a_{ij}					b_i	b_i^*
1							37/ 378	2825/ 27648
2	1/5	1/5					0	0
3	3/10	3/40	9/40				250/ 621	18575/ 48384
4	3/5	3/10	-9/10	6/5			125/ 594	13525/ 55296
5	1	-11/54	5/2	-70/27	35/27		0	277/ 14336
6	7/8	1631/ 55296	175/ 512	575/ 13824	44275/ 110592	253/ 4096	512/ 1771	1/4
j=		1	2	3	4	5		

See Numerical Recipes!

ORBITAL MECHANICS DEFINITIONS:

Consider the solar system, where the Sun effectively sits at the center of the system and the planets orbit about the Sun.



We are going to define a few useful quantities...

ORBITAL MECHANICS DEFINITIONS:

Perihelion distance, r_0:

Distance of closest approach in an elliptical orbit about the Sun

Aphelion distance, r_1:

Furthest distance in an elliptical orbit about the Sun

Semi-major axis of orbit *a*:

$2a = r_0 + r_1$

Eccentricity *e*:

Measure of non-circularity of an orbit

- $e < 1$: **ellipse**
- $e = 0$: **circle**
- $e = 1$: **parabola**
- $e > 1$: **hyperbola**

ORBITAL MECHANICS DEFINITIONS:

Relation between perihelion and aphelion distances:

$r_1 = r_0 (1+e)/(1-e)$

Velocity for a circular orbit:

acceleration $= v_c^2 / r = GM/r^2$

$v_c = (GM/r)^{1/2}$

Velocity at perihelion for an orbit with a given eccentricity:

$v_e = v_c(1+e)^{1/2}$, where v_c is evaluated at the perihelion r_0.

ORBITAL MECHANICS DEFINITIONS:

Solar Mass

$$1\,\mathrm{solar\,mass} = 1M_\odot = 1.98892 \times 10^{30}\mathrm{kg}$$

Astronomical Unit (Mean distance between Earth and Sun):

$$1\,\mathrm{AU} = 1.495979 \times 10^{11}\mathrm{m}$$

1 year (Earth orbital time)

$$1\,\mathrm{year} = 365.25\,\mathrm{days} = 3.15567 \times 10^{7}\mathrm{s}$$

ORBITAL MECHANICS DEFINITIONS:

Speed of light in solar system units

$$c = 2.997925 \times 10^8 \mathrm{m/s}$$

$$= 2.997925 \times 10^8 \mathrm{m/s} \left(\frac{1\mathrm{AU}}{1.495979 \times 10^{11}\mathrm{m}} \right) \left(\frac{3.15576 \times 10^7 \mathrm{s}}{1\mathrm{yr}} \right)$$

$$= 63421.1\,\mathrm{AU/yr}$$

MODELING OUR SOLAR SYSTEM

To make a model of our Solar System, we will begin by assuming all the bodies lie in a single plane. Hence, our system is two dimensional. We will only consider the effect of gravity:

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij}$$

In many cases, a single object provides the dominant force. In our Solar System, the Sun is by far the most important but Jupiter also provides interesting perturbative effects. For our models, we will generally restrict ourselves to considering the gravitational force provided by just a few key objects.

VARIABLE TIMESTEPS

We have already discussed the importance of variable timesteps, both in the context of many body systems and for ODE integration in general. In a Solar System model, it's essential.

Consider the orbits of Mercury and Pluto. Mercury whizzes around the Sun at approximately 48km/s, while Pluto drifts slowly at about 5km/s. So, for Mercury we need to use a relatively small timestep to accurately capture its orbit. For Pluto, a larger timestep would do nicely.

VARIABLE TIMESTEPS

What if you wanted to model Mercury and Pluto simultaneously?

What timestep should you use? If we use a short timestep appropriate for Mercury's orbit, Pluto will use many more timesteps than necessary. If we use a long timestep appropriate for Pluto's orbit, then Mercury's orbit will be inaccurate.

While the former choice (short timesteps) does not hurt the accuracy, it does hurt the run-time since we will spend a lot of time unnecessarily evolving Pluto's orbit.

Since run-time will become an increasingly important issue, it is important to develop schemes that increase computational efficiency.

VARIABLE TIMESTEPS

In short, variable timesteps are motivated by two main reasons:

1) Variable timesteps preserve a similar accuracy of integration for all the bodies in the system.

2) Variable timesteps use the computational resources most efficiently.

VARIABLE TIMESTEPS

In a variable timestepping scheme, each object has its own timestep. In principle, this is not difficult but it does increase the amount of bookkeeping in a code.

In particular, there are two issues that need to be solved:

1) How do we pick the timestep for each object?

2) We must evolve all the objects in a synchronous manner.

PICKING THE TIMESTEP

There are a number of ways to choose a timestep for an object. One useful scheme is to force each object to move no more than some fixed distance ε in a timestep Δt. In the case of Mercury and Pluto, this would result in roughly ten times as long a timestep for Pluto as for Mercury, since its velocity is one-tenth as large.

An appropriate timestep criterion might be something like:

$$\Delta t = \frac{\epsilon}{|\mathbf{v}|}$$

So long as |v| is constant, (as in a nearly circular orbit) this will work fine. But more generally this will fail when |v| gets very small but the acceleration remains large.

PICKING THE TIMESTEP

But what about acceleration? We can choose the minimum of:

$$\Delta t = \frac{\epsilon}{|\mathbf{v}|} \qquad \Delta t = \frac{\epsilon}{\sqrt{|\mathbf{a}|}}$$

This will be our approach.

How do we pick ε? A smaller value will result in a smaller timestep. Our choice is therefore determined by the accuracy, but in practice it is usually selected by trial and error. In the case of a softened gravitational force law, we can relate ε to the softening length.

SYNCHRONIZATION

Another issue that arises with variable timesteps when each object has its own timestep and each object is evolved at different rates.

This is not optimal because usually we need to compute the force on between various objects, or plot their positions, or compute energy conservation, etc., for a specific time for all objects.

In our Mercury and Pluto example, if we simply advanced Mercury and Pluto three timesteps from a fixed initial time, then Mercury will not be nearly as far advanced in time as Pluto since Pluto has a much longer timestep. What we would rather have is that Mercury takes *many* steps for a single step of Pluto, such that at the end of Pluto's step Mercury and Pluto end up at the same time.

We call this *synchronization.*

SYNCHRONIZATION

One method for performing such synchronization is to choose an overall large timestep for the entire system, typically taken to be a constant (but could be variable -- like the timestep of Pluto).

Within the large timestep, there is a loop over all the bodies in the system. For each body, you compute its own timestep using our criteria, and then have another loop that evolves howmany every timesteps that are needed to cover the full large timestep.

Since each object's timestep will not be an even multiple of the large timestep, you have to be careful not to exceed the large timestep on the last small timestep.

In the end there will be 3 nested loops:

1) The large timestep (main) loop
2) The loop over the number of bodies within 1)
3) The loop for each body's timestep within 2)

Verlet Equations

We could model position and velocity as follows:

$$x_{i+1} = x_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2 + O(\Delta t^3)$$

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1}) \Delta t + O(\Delta t^3)$$

These equations have the advantages that they are "self-starting", are second-order accurate, and easy to implement as long as the acceleration only depends on position (as is the case with gravity).

Unfortunately, for orbital motion the Verlet equations suffer from a small but serious deficiency....

LEAPFROG INTEGRATION

In the first few orbits, the motion determined by the Verlet equation is very close to the "real" motion. However, after many orbital periods the orbit will gradually diverge from the correct orbit.

This divergence can be slowed by reducing the timestep at the cost of computational time. But, in general, this scheme conserves energy and angular momentum poorly over large numbers of periods and is ill suited to the study of orbital motion.

Fortunately, there is an integration scheme that does not suffer from this gradually divergent behavior. While this scheme is also second-order accurate like the Verlet scheme, and therefore the error at each step is similar, on average the errors of this new scheme tend to average out rather than add coherently.

LEAPFROG INTEGRATION

We call this new scheme *leapfrog integration* for reasons that will become apparent, and it is a better choice for evolving systems over many dynamical times.

The leapfrog scheme has the advantage of being "symplectic", which means that the integrator is time-reversible. You can integrate the system forwards or backwards and arrive at the same ending or starting position.

For symplectic integration schemes, higher order errors tend to cancel out *on average* and hence such schemes maintain approximately the proper orbit forever.

LEAPFROG INTEGRATION

In the leapfrog scheme, the positions and velocities are "leapfrogged" over each other, with one being advanced between the full timesteps (e.g., 0, 1, 2, 3 ...) and the other being advanced between "halfsteps" (e.g., 1/2, 3/2, 5/2....). A full timestep thus progresses as follows:

$$x_{i+\frac{1}{2}} = x_i + \frac{1}{2} v_i \Delta t$$

$$v_{i+1} = v_i + a_{i+\frac{1}{2}} \Delta t$$

$$x_{i+1} = x_{i+\frac{1}{2}} + \frac{1}{2} v_{i+1} \Delta t$$

These equations are similar to the Verlet equations, but the acceleration at the half timestep is used to evolve the velocity. The order of accuracy is the same as the Verlet equations.

LEAPFROG INTEGRATION

One complication with the leapfrog method is that it is not precisely self-starting. The very first advance of position from x_0 to $x_{1/2}$ is only first-order accurate. Hence, if one uses the leapfrog scheme starting from t=0, the first halfstep is first-order accurate, and hence the entire calculation becomes first-order accurate!

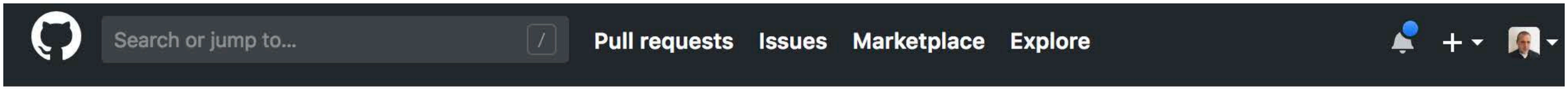
Fortunately, this can be easily remedied. To do so, the initialization of the position at the *very first* halfstep must be evolved according to a second-order accurate equation:

$$x_{i+\frac{1}{2}} = x_i + \frac{1}{2} v_i \Delta t + \frac{1}{4} a_i \Delta t^2 + O(\Delta t^3)$$

This is done *once*, and then you continue on with the leapfrog scheme.

Save Your Work

Make a GitHub project “astr-119-session-11”, and commit the programs my_first_jupyter_notebook.ipynb and test_matplotlib.ipynb you made today.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner **Repository name**

brantr / astr-119-session-5

Great repository names are short and memorable. Need inspiration? How about **fantastic-spork**.

Description (optional)

We learned a new trick! -- Jupyter notebooks.

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.