

**ASTR 119: Session 12**  
**Ordinary Differential Equations**  
**Coupled Systems**  
**Orbital Mechanics**

# Outline

- 1) Homework due 11/12 at 8:00am
- 2) Code Check In
- 3) Visualization of the Day
- 4) Coupled ordinary Differential Equations
- 5) Orbital Mechanics
- 6) Save your work to GitHub

# Homework, due Nov 12, 8:00am

1) Repeat the exercise from the 11/12 session, but use the Cash-Karp Runge-Kutta method with adaptive stepwise control

2) Evolve the system of equations:

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

Use the initial conditions  $y(x=0) = 0$  and  $dy/dx(x=0) = 1$ , 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 this week.**



# A Simple Coupled ODE

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

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

# A Simple Coupled ODE

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

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

With the boundary conditions of  $y(0) = 0$  and  $dy/dx(x_0) = 1$ , the solution is  $\sin(x)$ .

# A Simple Coupled ODE

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

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

With the boundary conditions of  $y(0) = 0$  and  $dy/dx(x_0) = 1$ , the solution is  $\sin(x)$ .

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

$$\begin{aligned}\frac{dy}{dx} &= z \\ \frac{dz}{dx} &= -y\end{aligned}$$

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



**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]: 1 def dydx(x,y):  
2  
3     #set the derivatives  
4  
5     # our equation is  $d^2y/dx^2 = -y$   
6  
7     #so we can write  
8     #dydx = z  
9     #dzdx = -y  
10  
11     #we will set y = y[0]  
12     #we will set z = y[1]  
13  
14     #declare an array  
15     y_derivs = np.zeros(2)  
16  
17     #set dydx = z  
18     y_derivs[0] = y[1]  
19  
20     #set dzdx = -y  
21     y_derivs[1] = -1*y[0]  
22  
23     #here we have to return an array  
24     return y_derivs
```

# Coupled ODE Solver

## Define the 4th order RK method

```
In [3]: 1 def rk4_mv_core(dydx,xi,yi,nv,h):
2
3     #declare k? arrays
4     k1 = np.zeros(nv)
5     k2 = np.zeros(nv)
6     k3 = np.zeros(nv)
7     k4 = np.zeros(nv)
8
9     #define x at 1/2 step
10    x_ipoh = xi + 0.5*h
11
12    #define x at 1 step
13    x_ipo = xi + h
14
15    #declare a temp y array
16    y_temp = np.zeros(nv)
17
18    #get k1 values
19    y_derivs = dydx(xi,yi)
20    k1[:] = h*y_derivs[:]
21
22    #get k2 values
23    y_temp[:] = yi[:] + 0.5*k1[:]
24    y_derivs = dydx(x_ipoh,y_temp)
25    k2[:] = h*y_derivs[:]
26
```

# Coupled ODE Solver

```
22  #get k2 values
23  y_temp[:] = yi[:] + 0.5*k1[:]
24  y_derivs = dydx(x_ipoh,y_temp)
25  k2[:] = h*y_derivs[:]
26
27  #get k3 values
28  y_temp[:] = yi[:] + 0.5*k2[:]
29  y_derivs = dydx(x_ipoh,y_temp)
30  k3[:] = h*y_derivs[:]
31
32  #get k4 values
33  y_temp[:] = yi[:] + k3[:]
34  y_derivs = dydx(x_ipo,y_temp)
35  k4[:] = h*y_derivs[:]
36
37
38  #advance y by a step h
39  yipo = yi + (k1 + 2*k2 + 2*k3 + k4)/6.
40
41  return yipo
```

(Cell continued)



# Coupled ODE Solver

## Define an adaptive step size driver for RK4

```
In [4]: 1  def rk4_mv_ad(dydx,x_i,y_i,nv,h,tol):
        2
        3      #define safety scale
        4      SAFETY      = 0.9
        5      H_NEW_FAC = 2.0
        6
        7      #set a maximum number of iterations
        8      imax = 10000
        9
        10     #set an iteration variable
        11     i = 0
        12
        13     #create an error
        14     Delta = np.full(nv,2*tol)
        15
        16     #remember the step
        17     h_step = h
        18
        19     #adjust step
        20     while(Delta.max()/tol > 1.0):
        21
```

# Coupled ODE Solver

```
19  #adjust step
20  while(Delta.max()/tol > 1.0):
21
22      #estimate our error by taking one step of size h
23      #vs. two steps of size h/2
24      y_2 = rk4_mv_core(dydx,x_i,y_i,nv,h_step)
25      y_1 = rk4_mv_core(dydx,x_i,y_i,nv,0.5*h_step)
26      y_11 = rk4_mv_core(dydx,x_i+0.5*h_step,y_1,nv,0.5*h_step)
27
28      #compute an error
29      Delta = np.fabs(y_2 - y_11)
30
31      #if the error is too large, take a smaller step
32      if(Delta.max()/tol > 1.0):
33
34          #our error is too large, decrease the step
35          h_step *= SAFETY * (Delta.max()/tol)**(-0.25)
36
37
38      #check iteration
39      if(i>=imax):
40          print("Too many iterations in rk4_mv_ad()")
41          raise StopIteration("Ending after i = ",i)
42
43      #iterate
44      i+=1
45
46
47      #next time, try to take a bigger step
48      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]: 1  def rk4_mv(dfdx,a,b,y_a,tol):
2
3      #dfdx is the derivative wrt x
4      #a is the lower bound
5      #b is the upper bound
6      #y_a are the boundary conditions
7      #tol is the tolerance for integrating y
8
9      #define our starting step
10     xi = a
11     yi = y_a.copy()
12
13     #an initial step size == make very small!
14     h = 1.0e-4 * (b-a)
15
16     #set a maximum number of iterations
17     imax = 10000
18
19     #set an iteration variable
20     i = 0
21
22     #set the number of coupled odes to the
23     #size of y_a
24     nv = len(y_a)
25
```



# Coupled ODE Solver

(Cell continued)

```
22  #set the number of coupled odes to the
23  #size of y_a
24  nv = len(y_a)
25
26
27  #set the initial conditions
28  x = np.full(1,a)
29  y = np.full((1,nv),y_a)
30
31  #set a flag
32  flag = 1
33
34  #loop until we reach the right side
35  while(flag):
36
37      #calculate y_i+1
38      yi_new, h_new, h_step = rk4_mv_ad(dydx,xi,yi,nv,h,tol)
39
40      #update the step
41      h = h_new
42
43      #prevent an overshoot
44      if(xi+h_step>b):
45
46          #take a smaller step
47          h = b-xi
48
49          #recalculate y_i+1
50          yi_new, h_new, h_step = rk4_mv_ad(dydx,xi,yi,nv,h,tol)
51
52          #break
53          flag = 0
54
```

# Coupled ODE Solver

(Cell continued)

```
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
```

# Coupled ODE Solver

## Perform the integration

In [6]:

```
1  a = 0.0
2  b = 2.0 * np.pi
3
4  y_0 = np.zeros(2)
5  y_0[0] = 0.0
6  y_0[1] = 1.0
7  nv = 2
8
9  tolerance = 1.0e-6
10
11  #perform the integration
12  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
```

```
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
```

(snip)

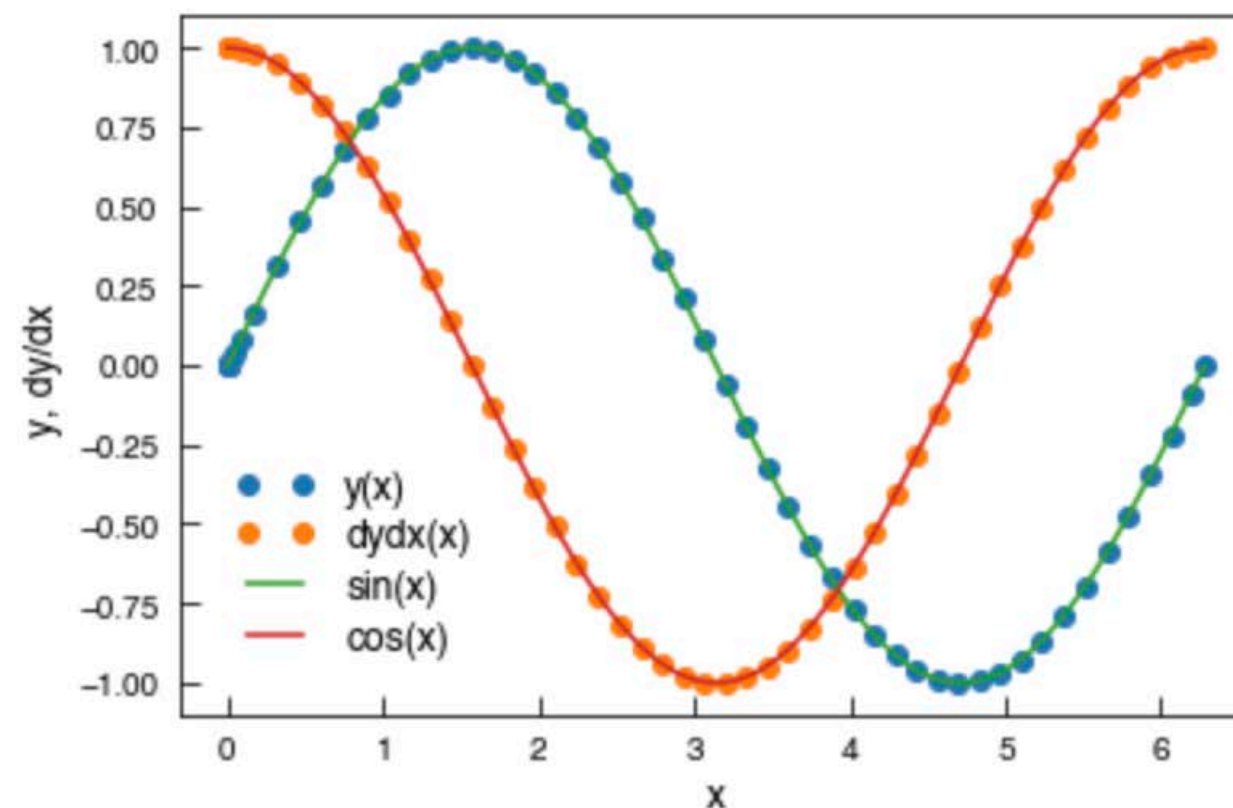


# Coupled ODE Solver

## 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>



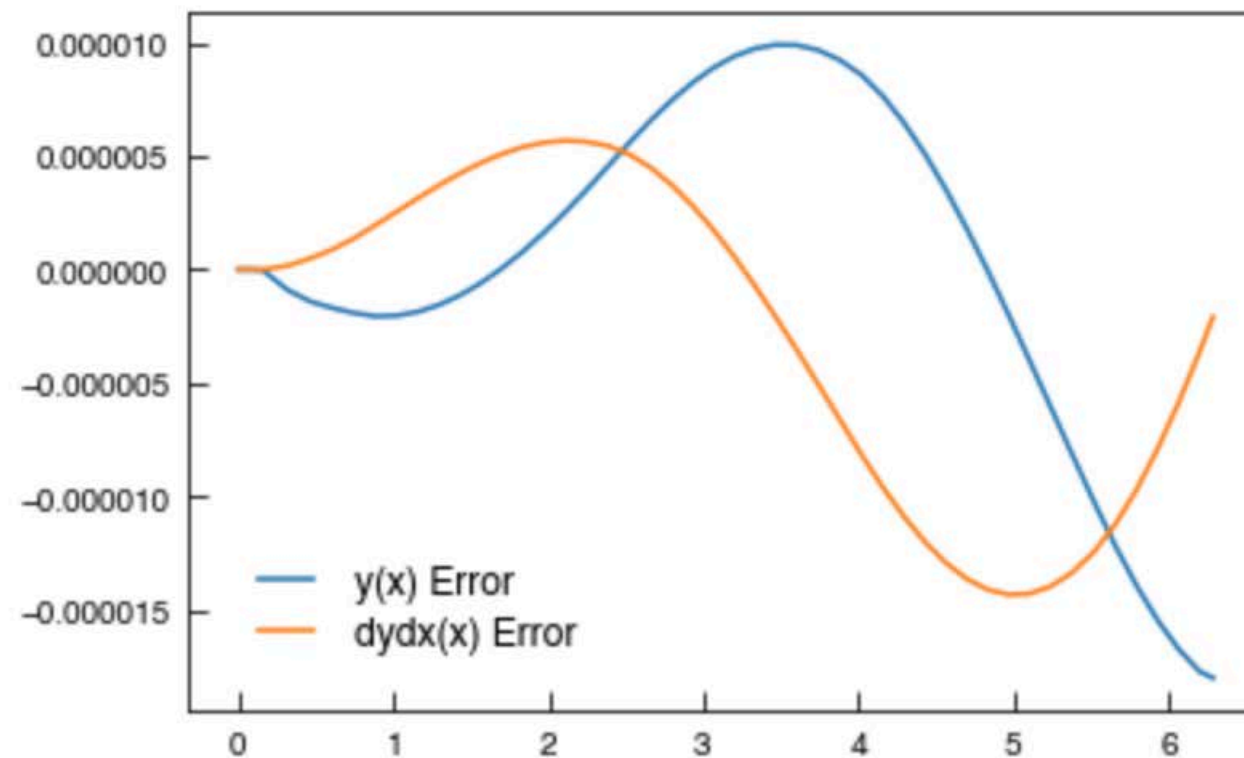
# Coupled ODE Solver

## Plot the error

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

```
In [8]: 1 sine = np.sin(x)
2 cosine = np.cos(x)
3
4 y_error = (y[:,0]-sine)
5 dydx_error = (y[:,1]-cosine)
6
7 plt.plot(x, y_error, label="y(x) Error")
8 plt.plot(x, dydx_error, label="dydx(x) Error")
9 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:

$$k_1 = hf(x_n, y_n) \qquad k_2 = hf(x_n + c_2h, y_n + a_{21}k_1)$$

$$k_3 = hf(x_n + c_3h, y_n + a_{31}k_1 + a_{32}k_2)$$

$$k_4 = hf(x_n + c_4h, y_n + a_{41}k_1 + a_{42}k_2 + a_{43}k_3)$$

$$k_5 = hf(x_n + c_5h, y_n + a_{51}k_1 + a_{52}k_2 + a_{53}k_3 + a_{54}k_4)$$

$$k_6 = hf(x_n + c_6h, y_n + a_{61}k_1 + a_{62}k_2 + a_{63}k_3 + a_{64}k_4 + a_{65}k_5)$$

$$y_{n+1} = y_n + b_1k_1 + b_2k_2 + b_3k_3 + b_4k_4 + b_5k_5 + b_6k_6 + O(h^6)$$

$$y_{n+1}^* = y_n + b_1^*k_1 + b_2^*k_2 + b_3^*k_3 + b_4^*k_4 + b_5^*k_5 + b_6^*k_6 + O(h^5)$$

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



# ADAPTIVE STEP-SIZE CONTROL

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

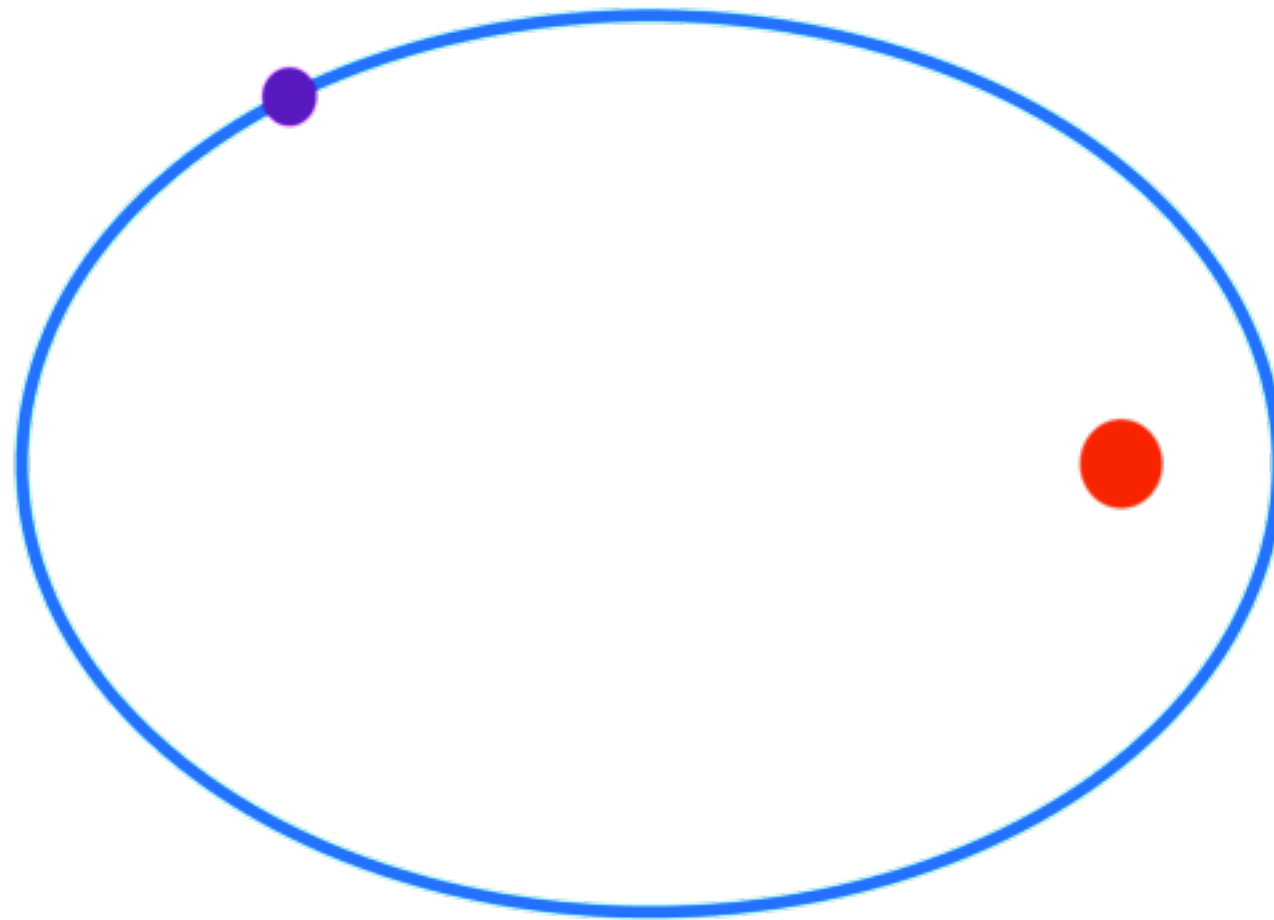
$i$	$C_i$	$a_{ij}$					$b_i$	$b_i^*$
1							$37/378$	$2825/27648$
2	$1/5$	$1/5$					0	0
3	$3/10$	$3/40$	$9/40$				$250/621$	$18575/48384$
4	$3/5$	$3/10$	$-9/10$	$6/5$			$125/594$	$13525/55296$
5	1	$-11/54$	$5/2$	$-70/27$	$35/27$		0	$277/14336$
6	$7/8$	$1631/55296$	$175/512$	$575/13824$	$44275/110592$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

$$\text{acceleration} = v_c^2 / r = GM/r^2$$

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

**Velocity at perihelion for an orbit with a given eccentricity:**

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

# ORBITAL MECHANICS

## DEFINITIONS:

### Solar Mass

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

### Astronomical Unit (Mean distance between Earth and Sun):

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

### 1 year (Earth orbital time)

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

# ORBITAL MECHANICS

## DEFINITIONS:

### Speed of light in solar system units

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

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

$$= 63421.1 \text{ AU/yr}$$

# 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  $\epsilon$  in a timestep  $\Delta t$ . 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  $|\mathbf{v}|$  is constant, (as in a nearly circular orbit) this will work fine. But more generally this will fail when  $|\mathbf{v}|$  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  $\epsilon$ ? 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  $\epsilon$  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:

$$\begin{aligned}x_{i+\frac{1}{2}} &= x_i + \frac{1}{2}v_i\Delta t \\v_{i+1} &= v_i + a_{i+\frac{1}{2}}\Delta t \\x_{i+1} &= x_{i+\frac{1}{2}} + \frac{1}{2}v_{i+1}\Delta t\end{aligned}$$

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-12”, and commit the programs `my_first_jupyter_notebook.ipynb` and `test_matplotlib.ipynb` you made today.



Search or jump to...



Pull requests

Issues

Marketplace

Explore



## 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.