Energy, Momentum and Conservation Laws

Morten Hjorth-Jensen[1,2]

Scott Pratt[1]

Carl Schmidt[3]

[1]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University, USA
[2]Department of Physics, University of Oslo, Norway
[3]Department of Physics and Astronomy, Michigan State University, USA

Feb 23, 2020

Work, Energy, Momentum and Conservation laws

Energy conservation is most convenient as a strategy for addressing problems where time does not appear. For example, a particle goes from position x_0 with speed v_0, to position x_f; what is its new speed? However, it can also be applied to problems where time does appear, such as in solving for the trajectory $x(t)$, or equivalently $t(x)$.

Before we start formulating a strategy for energy conservation, we need to discuss integration methods and the concept of work and how it relates to energy.

Work and Energy

Till our own material is placed here, we recommend reading chapters 10-14 of Malthe-Sørenssen and "Taylor chapters 3 and 4":" `https://www.uscibooks.com/taylor2.htm`."

On work, chapter 10 of Malthe-Sørenssen is a good read.

Energy Conservation

Energy is conserved in the case where the potential energy, $V(\boldsymbol{r})$, depends only on position, and not on time. The force is determined by V,

$$\boldsymbol{F}(\boldsymbol{r}) = -\nabla V(\boldsymbol{r}). \tag{1}$$

The net energy, $E = V + K$ where K is the kinetic energy, is then conserved,

$$\begin{aligned}\frac{d}{dt}(K+V) &= \frac{d}{dt}\left(\frac{m}{2}(v_x^2+v_y^2+v_z^2)+V(\boldsymbol{r})\right) \\ &= m\left(v_x\frac{dv_x}{dt}+v_y\frac{dv_y}{dt}+v_z\frac{dv_z}{dt}\right)+\partial_x V\frac{dx}{dt}+\partial_y V\frac{dy}{dt}+\partial_z V\frac{dz}{dt} \\ &= v_xF_x+v_yF_y+v_zF_z-F_xv_x-F_yv_y-F_zv_z=0.\end{aligned} \tag{2}$$

The same proof can be written more compactly with vector notation,

$$\begin{aligned}\frac{d}{dt}\left(\frac{m}{2}v^2+V(\boldsymbol{r})\right) &= m\boldsymbol{v}\cdot\dot{\boldsymbol{v}}+\nabla V(\boldsymbol{r})\cdot\dot{\boldsymbol{r}} \\ &= \boldsymbol{v}\cdot\boldsymbol{F}-\boldsymbol{F}\cdot\boldsymbol{v}=0.\end{aligned} \tag{3}$$

Inverting the expression for kinetic energy,

$$v=\sqrt{2K/m}=\sqrt{2(E-V)/m}, \tag{4}$$

allows one to solve for the one-dimensional trajectory $x(t)$, by finding $t(x)$,

$$t=\int_{x_0}^{x}\frac{dx'}{v(x')}=\int_{x_0}^{x}\frac{dx'}{\sqrt{2(E-V(x'))/m}}. \tag{5}$$

Note this would be much more difficult in higher dimensions, because you would have to determine which points, x, y, z, the particles might reach in the trajectory, whereas in one dimension you can typically tell by simply seeing whether the kinetic energy is positive at every point between the old position and the new position.

Consider a simple harmonic oscillator potential, $V(x)=kx^2/2$, with a particle emitted from $x=0$ with velocity v_0. Solve for the trajectory $t(x)$,

$$\begin{aligned}t &= \int_0^x\frac{dx'}{\sqrt{2(E-kx^2/2)/m}} \\ &= \sqrt{m/k}\int_0^x\frac{dx'}{\sqrt{x_{\rm max}^2-x'^2}}, \quad x_{\rm max}^2=2E/k.\end{aligned} \tag{6}$$

Here $E=mv_0^2/2$ and $x_{\rm max}$ is defined as the maximum displacement before the particle turns around. This integral is done by the substitution $\sin\theta=x/x_{\rm max}$.

$$\begin{aligned}(k/m)^{1/2}t &= \sin^{-1}(x/x_{\rm max}), \\ x &= x_{\rm max}\sin\omega t, \quad \omega=\sqrt{k/m}.\end{aligned} \tag{7}$$

Numerical Integration

As an example of an integral to solve numerically, consider the following integral. First, rewrite the integral as a sum,

$$t = \sum_{n=1}^{N} \Delta x[2(E - V(x_n)/m]^{-1/2}, \tag{8}$$

where $\Delta x = (x - x_0)/N$ and $x_n = x_0 + (n - 1/2)\Delta x$. Note that for best accuracy the value of x_n has been placed in the center of the n^{th} interval. The accuracy will improve for higher values of N, or equivalently, smaller step size Δx.

Rectangular Rule

```
from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the Rectangular rule

for (j = 0; j <= n; j++){
         x = (j+0.5)*step+;   // midpoint of a given rectangle
         RectangleSum+=(*func)(x);   //  add value of function.
      }
      RectangleSum *= step;  //  multiply with step length.
      return RectangleSum;


def Rectangular(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a+h*0.5
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = h*s
   return h*s
#  function to compute pi
def function(x):
    return x*x
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
```

```
exact = integrate(function(x), (x, a, b))
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Rectangular(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```

Trapezoidal Rule

```
from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the trapezoidal rule
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s
#  function to compute pi
def function(x):
    return x*x
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
exact = integrate(function(x), (x, a, b))
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Trapez(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```

Simpsons' rule

```
def trapezoidal(self, a,b, n, func):
        """
        Integrate the function func
        using the trapezoidal rule from a to b,
        with n points. Returns value from numerical integration
        """
        step = (b-a)/float(n)

        sum = func(a)/float(2);
        for i in xrange(1,n):
            sum = sum + func(a+i*step)
        sum = sum + func(b)/float(2)

        return sum*step

    def simpson(self,a,b,n,func):
        """Same as trapezoidal, but use simpsons rule"""
        step = (b-a)/float(n)

        sum = func(a)/float(2);
        for i in xrange(1,n):
            sum = sum + func(a+i*step)*(3+(-1)**(i+1))
        sum = sum + func(b)/float(2)

        return sum*step/3.0


from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the Rectangular rule
def Simpson(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a+h*0.5
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = h*s
   return h*s
#  function to compute pi
def function(x):
    return x*x
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
exact = integrate(function(x), (x, a, b))
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Simpson(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```

Conservation of Momentum

Newton's third law which we met earlier states that **For every action there is an equal and opposite reaction**, is more accurately stated as **If two bodies exert forces on each other, these forces are equal in magnitude and opposite in direction**.

This means that for two bodies i and j, if the force on i due to j is called $\boldsymbol{F}_{ij}$, then

$$\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}. \tag{9}$$

Newton's second law, $\boldsymbol{F} = m\boldsymbol{a}$, can be written for a particle i as

$$\boldsymbol{F}_i = \sum_{j\neq i} \boldsymbol{F}_{ij} = m_i\boldsymbol{a}_i, \tag{10}$$

where $\boldsymbol{F}_i$ (a single subscript) denotes the net force acting on i. Because the mass of i is fixed, one can see that

$$\boldsymbol{F}_i = \frac{d}{dt} m_i \boldsymbol{v}_i = \sum_{j \neq i} \boldsymbol{F}_{ij}. \tag{11}$$

Now, one can sum over all the particles and obtain

$$\begin{aligned} \frac{d}{dt} \sum_i m_i v_i &= \sum_{ij, i\neq j} \boldsymbol{F}_{ij} \\ &= 0. \end{aligned} \tag{12}$$

The last step made use of the fact that for every term ij, there is an equivalent term ji with opposite force. Because the momentum is defined as $m\boldsymbol{v}$, for a system of particles,

$$\frac{d}{dt} \sum_i m_i \boldsymbol{v}_i = 0, \quad \text{for isolated particles.} \tag{13}$$

By "isolated" one means that the only force acting on any particle i are those originating from other particles in the sum, i.e. “no external” forces. Thus, Newton’s third law leads to the conservation of total momentum,

$$\begin{aligned} \boldsymbol{P} &= \sum_i m_i \boldsymbol{v}_i, \\ \frac{d}{dt}\boldsymbol{P} &= 0. \end{aligned} \tag{14}$$

Consider the rocket of mass M moving with velocity v. After a brief instant, the velocity of the rocket is $v + \Delta v$ and the mass is $M - \Delta M$. Momentum conservation gives

$$\begin{aligned} Mv &= (M - \Delta M)(v + \Delta v) + \Delta M(v - v_e) \\ 0 &= -\Delta M v + M\Delta v + \Delta M(v - v_e), \\ 0 &= M\Delta v - \Delta M v_e. \end{aligned}$$

In the second step we ignored the term $\Delta M \Delta v$ because it is doubly small. The last equation gives

$$\begin{aligned} \Delta v &= \frac{v_e}{M} \Delta M, \\ \frac{dv}{dt} &= \frac{v_e}{M} \frac{dM}{dt}. \end{aligned} \tag{15}$$

Integrating the expression with lower limits $v_0 = 0$ and M_0, one finds

$$\begin{aligned} v &= v_e \int_{M_0}^{M} \frac{dM'}{M'} \\ v &= -v_e \ln(M/M_0) \\ &= -v_e \ln[(M_0 - \alpha t)/M_0]. \end{aligned}$$

Because the total momentum of an isolated system is constant, one can also quickly see that the center of mass of an isolated system is also constant. The center of mass is the average position of a set of masses weighted by the mass,

$$\bar{x} = \frac{\sum_i m_i x_i}{\sum_i m_i}. \tag{16}$$

The rate of change of $\bar{x}$ is

$$\dot{\bar{x}} = \frac{1}{M} \sum_i m_i \dot{x}_i = \frac{1}{M} P_x. \tag{17}$$

Thus if the total momentum is constant the center of mass moves at a constant velocity, and if the total momentum is zero the center of mass is fixed.

Conservation of Angular Momentum

Consider a case where the force always points radially,

$$\boldsymbol{F}(\boldsymbol{r}) = F(r)\hat{r}, \tag{18}$$

where $\hat{r}$ is a unit vector pointing outward from the origin. The angular momentum is defined as

$$\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p} = m\boldsymbol{r} \times \boldsymbol{v}. \tag{19}$$

The rate of change of the angular momentum is

$$\begin{aligned} \frac{d\boldsymbol{L}}{dt} &= m\boldsymbol{v} \times \boldsymbol{v} + m\boldsymbol{r} \times \dot{\boldsymbol{v}} \qquad (20) \\ &= m\boldsymbol{v} \times \boldsymbol{v} + \boldsymbol{r} \times \boldsymbol{F} = 0. \end{aligned}$$

The first term is zero because $\boldsymbol{v}$ is parallel to itself, and the second term is zero because $\boldsymbol{F}$ is parallel to $\boldsymbol{r}$.

As an aside, one can see from the Levi-Civita symbol that the cross product of a vector with itself is zero. Here, we consider a vector

$$\begin{aligned} \boldsymbol{V} &= \boldsymbol{A} \times \boldsymbol{A}, \qquad (21) \\ V_i &= (\boldsymbol{A} \times \boldsymbol{A})_i = \sum_{jk} \epsilon_{ijk} A_j A_k. \end{aligned}$$

For any term i, there are two contributions. For example, for i denoting the x direction, either j denotes the y direction and k denotes the z direction, or vice versa, so

$$V_1 = \epsilon_{123} A_2 A_3 + \epsilon_{132} A_3 A_2. \tag{22}$$

This is zero by the antisymmetry of ϵ under permutations.

If the force is not radial, $\boldsymbol{r} \times \boldsymbol{F} \neq 0$ as above, and angular momentum is no longer conserved,

$$\frac{d\boldsymbol{L}}{dt} = \boldsymbol{r} \times \boldsymbol{F} \equiv \boldsymbol{\tau}, \tag{23}$$

where $\boldsymbol{\tau}$ is the torque.

For a system of isolated particles, one can write

$$\begin{aligned} \frac{d}{dt} \sum_i \boldsymbol{L}_i &= \sum_{i \neq j} \boldsymbol{r}_i \times \boldsymbol{F}_{ij} \\ &= \frac{1}{2} \sum_{i \neq j} \boldsymbol{r}_i \times \boldsymbol{F}_{ij} + \boldsymbol{r}_j \times \boldsymbol{F}_{ji} \\ &= \frac{1}{2} \sum_{i \neq j} (\boldsymbol{r}_i - \boldsymbol{r}_j) \times \boldsymbol{F}_{ij} = 0, \end{aligned} \tag{24}$$

where the last step used Newton's third law, $\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji}$. If the forces between the particles are radial, i.e. $\boldsymbol{F}_{ij} \mid\mid (\boldsymbol{r}_i - \boldsymbol{r}_j)$, then each term in the sum is zero and the net angular momentum is fixed. Otherwise, you could imagine an isolated system that would start spinning spontaneously.

One can write the torque about a given axis, which we will denote as $\hat{z}$, in polar coordinates, where

$$x = r \sin\theta \cos\phi, \quad y = r \sin\theta \cos\phi, \quad z = r \cos\theta, \tag{25}$$

to find the z component of the torque,

$$\begin{aligned} \tau_z &= x F_y - y F_x \\ &= -r \sin\theta \left\{ \cos\phi \partial_y - \sin\phi \partial_x \right\} V(x, y, z). \end{aligned} \tag{26}$$

One can use the chain rule to write the partial derivative w.r.t. ϕ (keeping r and θ fixed),

$$\begin{aligned} \partial_\phi &= \frac{\partial x}{\partial \phi} \partial_x + \frac{\partial_y}{\partial \phi} \partial_y + \frac{\partial z}{\partial \phi} \partial_z \\ &= -r \sin\theta \sin\phi \partial_x + \sin\theta \cos\phi \partial_y. \end{aligned} \tag{27}$$

Combining the two equations,

$$\tau_z = -\partial_\phi V(r,\theta,\phi). \tag{28}$$

Thus, if the potential is independent of the azimuthal angle ϕ, there is no torque about the z axis and L_z is conserved.

Symmetries and Conservation Laws

When we derived the conservation of energy, we assumed that the potential depended only on position, not on time. If it depended explicitly on time, one can quickly see that the energy would have changed at a rate $\partial_t V(x,y,z,t)$. Note that if there is no explicit dependence on time, i.e. $V(x,y,z)$, the potential energy can depend on time through the variations of x,y,z with time. However, that variation does not lead to energy non-conservation. Further, we just saw that if a potential does not depend on the azimuthal angle about some axis, ϕ, that the angular momentum about that axis is conserved.

Now, we relate momentum conservation to translational invariance. Considering a system of particles with positions, $\boldsymbol{r}_i$, if one changed the coordinate system by a translation by a differential distance $\boldsymbol{\epsilon}$, the net potential would change by

$$\begin{aligned}\delta V(\boldsymbol{r}_1,\boldsymbol{r}_2\cdots) &= \sum_i \boldsymbol{\epsilon}\cdot\nabla_i V(\boldsymbol{r}_1,\boldsymbol{r}_2,\cdots) \\ &= -\sum_i \boldsymbol{\epsilon}\cdot\boldsymbol{F}_i \\ &= -\frac{d}{dt}\sum_i \boldsymbol{\epsilon}\cdot\boldsymbol{p}_i.\end{aligned} \tag{29}$$

Thus, if the potential is unchanged by a translation of the coordinate system, the total momentum is conserved. If the potential is translationally invariant in a given direction, defined by a unit vector, $\hat{\epsilon}$ in the $\boldsymbol{\epsilon}$ direction, one can see that

$$\hat{\epsilon}\cdot\nabla_i V(\boldsymbol{r}_i) = 0. \tag{30}$$

The component of the total momentum along that axis is conserved. This is rather obvious for a single particle. If $V(\boldsymbol{r})$ does not depend on some coordinate x, then the force in the x direction is $F_x = -\partial_x V = 0$, and momentum along the x direction is constant.

We showed how the total momentum of an isolated system of particle was conserved, even if the particles feel internal forces in all directions. In that case the potential energy could be written

$$V = \sum_{i,j\le i} V_{ij}(\boldsymbol{r}_i - \boldsymbol{r}_j). \tag{31}$$

In this case, a translation leads to $\boldsymbol{r}_i \rightarrow \boldsymbol{r}_i + \boldsymbol{\epsilon}$, with the translation equally affecting the coordinates of each particle. Because the potential depends only on the relative coordinates, δV is manifestly zero. If one were to go through the exercise of calculating δV for small $\boldsymbol{\epsilon}$, one would find that the term $\nabla_i V(\boldsymbol{r}_i - \boldsymbol{r}_j)$ would be canceled by the term $\nabla_j V(\boldsymbol{r}_i - \boldsymbol{r}_j)$.

The relation between symmetries of the potential and conserved quantities (also called constants of motion) is one of the most profound concepts one should gain from this course. It plays a critical role in all fields of physics. This is especially true in quantum mechanics, where a quantity A is conserved if its operator commutes with the Hamiltonian. For example if the momentum operator $-i\hbar\partial_x$ commutes with the Hamiltonian, momentum is conserved, and clearly this operator commutes if the Hamiltonian (which represents the total energy, not just the potential) does not depend on x. Also in quantum mechanics the angular momentum operator is $L_z = -i\hbar\partial_\phi$. In fact, if the potential is unchanged by rotations about some axis, angular momentum about that axis is conserved. We return to this concept, from a more formal perspective, later in the course when Lagrangian mechanics is presented.

Bulding a code for the Earth-Sun system

We will now venture into a study of a system which is energy conserving. The aim is to see if we (since it is not possible to solve the general equations analytically) we can develop stable numerical algorithms whose results we can trust!

We solve the equations of motion numerically. We will also compute quantities like the energy numerically.

We start with a simpler case first, the Earth-Sun system in two dimensions only. The gravitational force F_G on the earth from the sun is

$$\boldsymbol{F}_G = -\frac{GM_\odot M_E}{r^3}\boldsymbol{r},$$

where G is the gravitational constant,

$$M_E = 6\times 10^{24}\mathrm{Kg},$$

the mass of Earth,

$$M_\odot = 2\times 10^{30}\mathrm{Kg},$$

the mass of the Sun and

$$r = 1.5\times 10^{11}\mathrm{m},$$

is the distance between Earth and the Sun. The latter defines what we call an astronomical unit **AU**. From Newton's second law we have then for the x direction

$$\frac{d^2x}{dt^2} = -\frac{F_x}{M_E},$$

and

$$\frac{d^2y}{dt^2} = -\frac{F_y}{M_E},$$

for the y direction.

Here we will use that $x = r\cos(\theta)$, $y = r\sin(\theta)$ and

$$r = \sqrt{x^2 + y^2}.$$

We can rewrite

$$F_x = -\frac{GM_\odot M_E}{r^2}\cos(\theta) = -\frac{GM_\odot M_E}{r^3}x,$$

and

$$F_y = -\frac{GM_\odot M_E}{r^2}\sin(\theta) = -\frac{GM_\odot M_E}{r^3}y,$$

for the y direction.

We can rewrite these two equations

$$F_x = -\frac{GM_\odot M_E}{r^2}\cos(\theta) = -\frac{GM_\odot M_E}{r^3}x,$$

and

$$F_y = -\frac{GM_\odot M_E}{r^2}\sin(\theta) = -\frac{GM_\odot M_E}{r^3}y,$$

as four first-order coupled differential equations

$$\frac{dv_x}{dt} = -\frac{GM_\odot}{r^3}x,$$
$$\frac{dx}{dt} = v_x,$$
$$\frac{dv_y}{dt} = -\frac{GM_\odot}{r^3}y,$$
$$\frac{dy}{dt} = v_y.$$

Building a code for the solar system, final coupled equations

The four coupled differential equations

$$\frac{dv_x}{dt} = -\frac{GM_\odot}{r^3}x,$$
$$\frac{dx}{dt} = v_x,$$
$$\frac{dv_y}{dt} = -\frac{GM_\odot}{r^3}y,$$
$$\frac{dy}{dt} = v_y,$$

can be turned into dimensionless equations or we can introduce astronomical units with 1 AU $= 1.5 \times 10^{11}$.

Using the equations from circular motion (with $r = 1\mathrm{AU}$)

$$\frac{M_E v^2}{r} = F = \frac{GM_\odot M_E}{r^2},$$

we have

$$GM_\odot = v^2 r,$$

and using that the velocity of Earth (assuming circular motion) is $v = 2\pi r/\mathrm{yr} = 2\pi\mathrm{AU}/\mathrm{yr}$, we have

$$GM_\odot = v^2 r = 4\pi^2 \frac{(\mathrm{AU})^3}{\mathrm{yr}^2}.$$

Building a code for the solar system, discretized equations

The four coupled differential equations can then be discretized using Euler's method as (with step length h)

$$v_{x,i+1} = v_{x,i} - h\frac{4\pi^2}{r_i^3}x_i,$$

$$x_{i+1} = x_i + hv_{x,i},$$

$$v_{y,i+1} = v_{y,i} - h\frac{4\pi^2}{r_i^3}y_i,$$

$$y_{i+1} = y_i + hv_{y,i},$$

Code Example with Euler's Method

The code here implements Euler's method for the Earth-Sun system using a more compact way of representing the vectors. Alternatively, you could have spelled out all the variables v_x, v_y, x and y as one-dimensional arrays.

```
# Common imports
import numpy as np
import pandas as pd
from math import *
import matplotlib.pyplot as plt
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)
```

```
if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')


DeltaT = 0.001
#set up arrays
tfinal = 10 # in years
n = ceil(tfinal/DeltaT)
# set up arrays for t, a, v, and x
t = np.zeros(n)
v = np.zeros((n,2))
r = np.zeros((n,2))
# Initial conditions as compact 2-dimensional arrays
r0 = np.array([1.0,0.0])
v0 = np.array([0.0,2*pi])
r[0] = r0
v[0] = v0
Fourpi2 = 4*pi*pi
# Start integrating using Euler's method
for i in range(n-1):
    # Set up the acceleration
    # Here you could have defined your own function for this
    rabs = sqrt(sum(r[i]*r[i]))
    a =  -Fourpi2*r[i]/(rabs**3)
    # update velocity, time and position using Euler's forward method
    v[i+1] = v[i] + DeltaT*a
    r[i+1] = r[i] + DeltaT*v[i]
    t[i+1] = t[i] + DeltaT
# Plot position as function of time
fig, ax = plt.subplots()
#ax.set_xlim(0, tfinal)
ax.set_ylabel('x[m]')
ax.set_xlabel('y[m]')
ax.plot(r[:,0], r[:,1])
```

```
fig.tight_layout()
save_fig("EarthSunEuler")
plt.show()
```

Problems with Euler's Method

We notice here that Euler's method doesn't give a stable orbit. It means that we cannot trust Euler's method. In a deeper way, as we will see in homework 5, Euler's method does not conserve energy. It is an example of an integrator which is not symplectic.

Here we present thus two methods, which with simple changes allow us to avoid these pitfalls. The simplest possible extension is the so-called Euler-Cromer method. The changes we need to make to our code are indeed marginal here. We need simply to replace

```
    r[i+1] = r[i] + DeltaT*v[i]
```

in the above code with the velocity at the new time t_{i+1}

```
    r[i+1] = r[i] + DeltaT*v[i+1]
```

By this simple caveat we get stable orbits. Below we derive the Euler-Cromer method as well as one of the most utlized algorithms for sovling the above type of problems, the so-called Velocity-Verlet method.

Deriving the Euler-Cromer Method

Let us repeat Euler's method. We have a differential equation

$$y'(t_i) = f(t_i, y_i) \tag{32}$$

and if we truncate at the first derivative, we have from the Taylor expansion

$$y_{i+1} = y(t_i) + (\Delta t) f(t_i, y_i) + O(\Delta t^2), \tag{33}$$

which when complemented with $t_{i+1} = t_i + \Delta t$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(\Delta t^2)$, however the total error is the sum over all steps $N = (b-a)/(\Delta t)$ for $t \in [a, b]$, yielding thus a global error which goes like $NO(\Delta t^2) \approx O(\Delta t)$.

To make Euler's method more precise we can obviously decrease Δt (increase N), but this can lead to loss of numerical precision. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like.

Euler's method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at y_1 using the velocity at v_0. A simple variation is to determine x_{n+1} using the velocity at v_{n+1}, that is (in a slightly more generalized form)

$$y_{n+1} = y_n + v_{n+1} + O(\Delta t^2) \tag{34}$$

and

$$v_{n+1} = v_n + (\Delta t)a_n + O(\Delta t^2). \tag{35}$$

The acceleration a_n is a function of $a_n(y_n, v_n, t_n)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Exercise: go back to the above code with Euler's method and add the Euler-Cromer method.

Deriving the Velocity-Verlet Method

Let us stay with x (position) and v (velocity) as the quantities we are interested in.

We have the Taylor expansion for the position given by

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

The corresponding expansion for the velocity is

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

Via Newton's second law we have normally an analytical expression for the derivative of the velocity, namely

$$a_i = \frac{d^2x}{dt^2}|_i = \frac{dv}{dt}|_i = \frac{F(x_i, v_i, t_i)}{m}.$$

If we add to this the corresponding expansion for the derivative of the velocity

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

and retain only terms up to the second derivative of the velocity since our error goes as $O(h^3)$, we have

$$(\Delta t)v_i^{(2)} \approx a_{i+1} - a_i.$$

We can then rewrite the Taylor expansion for the velocity as

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

The velocity Verlet method

Our final equations for the position and the velocity become then

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

and

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

Note well that the term a_{i+1} depends on the position at x_{i+1}. This means that you need to calculate the position at the updated time t_{i+1} before the computing the next velocity. Note also that the derivative of the velocity at the time t_i used in the updating of the position can be reused in the calculation of the velocity update as well.

Adding the Velocity-Verlet Method

We can now easily add the Verlet method to our original code as

```
DeltaT = 0.01
#set up arrays
tfinal = 10
n = ceil(tfinal/DeltaT)
# set up arrays for t, a, v, and x
t = np.zeros(n)
v = np.zeros((n,2))
r = np.zeros((n,2))
# Initial conditions as compact 2-dimensional arrays
r0 = np.array([1.0,0.0])
v0 = np.array([0.0,2*pi])
r[0] = r0
v[0] = v0
Fourpi2 = 4*pi*pi
# Start integrating using the Velocity-Verlet  method
for i in range(n-1):
    # Set up forces, air resistance FD, note now that we need the norm of the vecto
    # Here you could have defined your own function for this
    rabs = sqrt(sum(r[i]*r[i]))
    a =  -Fourpi2*r[i]/(rabs**3)
    # update velocity, time and position using the Velocity-Verlet method
    r[i+1] = r[i] + DeltaT*v[i]+0.5*(DeltaT**2)*a
    rabs = sqrt(sum(r[i+1]*r[i+1]))
    anew = -4*(pi**2)*r[i+1]/(rabs**3)
    v[i+1] = v[i] + 0.5*DeltaT*(a+anew)
    t[i+1] = t[i] + DeltaT
# Plot position as function of time
fig, ax = plt.subplots()
ax.set_ylabel('x[m]')
ax.set_xlabel('y[m]')
ax.plot(r[:,0], r[:,1])
fig.tight_layout()
save_fig("EarthSunVV")
plt.show()
```

You can easily generalize the calculation of the forces by defining a function which takes in as input the various variables. We leave this as a challenge to you.

Studying Energy Conservation

In order to study the conservation of energy, we will need to perform a numerical integration, unless we can integrate analytically. Here we present the Trapezoidal rule as a the simplest possible approximation.

The integral

$$I = \int_a^b f(x)dx \tag{36}$$

has a very simple meaning. The integral is the area enscribed by the function $f(x)$ starting from $x = a$ to $x = b$. It is subdivided in several smaller areas whose evaluation is to be approximated by different techniques. The areas under the curve can for example be approximated by rectangular boxes or trapezoids.

In considering equal step methods, our basic approach is that of approximating a function $f(x)$ with a polynomial of at most degree $N - 1$, given N integration points. If our polynomial is of degree 1, the function will be approximated with $f(x) \approx a_0 + a_1 x$.

The algorithm for these integration methods is rather simple, and the number of approximations perhaps unlimited!

- Choose a step size $h = (b - a)/N$ where N is the number of steps and a and b the lower and upper limits of integration.

- With a given step length we rewrite the integral as

$$\int_a^b f(x)dx = \int_a^{a+h} f(x)dx + \int_{a+h}^{a+2h} f(x)dx + \dots \int_{b-h}^{b} f(x)dx.$$

- The strategy then is to find a reliable polynomial approximation for $f(x)$ in the various intervals. Choosing a given approximation for $f(x)$, we obtain a specific approximation to the integral.

- With this approximation to $f(x)$ we perform the integration by computing the integrals over all subintervals.

One possible strategy then is to find a reliable polynomial expansion for $f(x)$ in the smaller subintervals. Consider for example evaluating

$$\int_a^{a+2h} f(x)dx,$$

which we rewrite as

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0+h} f(x)dx. \tag{37}$$

We have chosen a midpoint x_0 and have defined $x_0 = a + h$.

Using Lagrange's interpolation formula

$$P_N(x) = \sum_{i=0}^{N} \prod_{k\neq i} \frac{x - x_k}{x_i - x_k} y_i,$$

we could attempt to approximate the function $f(x)$ with a first-order polynomial in x in the two sub-intervals $x \in [x_0 - h, x_0]$ and $x \in [x_0, x_0 + h]$. A first order polynomial means simply that we have for say the interval $x \in [x_0, x_0 + h]$

$$f(x) \approx P_1(x) = \frac{x - x_0}{(x_0 + h) - x_0} f(x_0 + h) + \frac{x - (x_0 + h)}{x_0 - (x_0 + h)} f(x_0),$$

and for the interval $x \in [x_0 - h, x_0]$

$$f(x) \approx P_1(x) = \frac{x - (x_0 - h)}{x_0 - (x_0 - h)} f(x_0) + \frac{x - x_0}{(x_0 - h) - x_0} f(x_0 - h).$$

Having performed this subdivision and polynomial approximation, one from $x_0 - h$ to x_0 and the other from x_0 to $x_0 + h$,

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0} f(x)dx + \int_{x_0}^{x_0+h} f(x)dx,$$

we can easily calculate for example the second integral as

$$\int_{x_0}^{x_0+h} f(x)dx \approx \int_{x_0}^{x_0+h} \left(\frac{x - x_0}{(x_0 + h) - x_0} f(x_0 + h) + \frac{x - (x_0 + h)}{x_0 - (x_0 + h)} f(x_0) \right) dx.$$

This integral can be simplified to

$$\int_{x_0}^{x_0+h} f(x)dx \approx \int_{x_0}^{x_0+h} \left(\frac{x - x_0}{h} f(x_0 + h) - \frac{x - (x_0 + h)}{h} f(x_0) \right) dx,$$

resulting in

$$\int_{x_0}^{x_0+h} f(x)dx = \frac{h}{2} \left(f(x_0 + h) + f(x_0) \right) + O(h^3).$$

Here we added the error made in approximating our integral with a polynomial of degree 1.

The other integral gives

$$\int_{x_0-h}^{x_0} f(x)dx = \frac{h}{2} \left(f(x_0) + f(x_0 - h) \right) + O(h^3),$$

and adding up we obtain

$$\int_{x_0-h}^{x_0+h} f(x)dx = \frac{h}{2} \left(f(x_0 + h) + 2f(x_0) + f(x_0 - h) \right) + O(h^3), \tag{38}$$

which is the well-known trapezoidal rule. Concerning the error in the approximation made, $O(h^3) = O((b-a)^3/N^3)$, you should note that this is the local error. Since we are splitting the integral from a to b in N pieces, we will have to perform approximately N such operations.

This means that the *global error* goes like $\approx O(h^2)$. The trapezoidal reads then

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right), \tag{39}$$

with a global error which goes like $O(h^2)$.

Hereafter we use the shorthand notations $f_{-h} = f(x_0 - h)$, $f_0 = f(x_0)$ and $f_h = f(x_0 + h)$.

The correct mathematical expression for the local error for the trapezoidal rule is

$$\int_a^b f(x)dx - \frac{b-a}{2}\left[f(a) + f(b)\right] = -\frac{h^3}{12} f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - T_h(f) = -\frac{b-a}{12} h^2 f^{(2)}(\xi),$$

where T_h is the trapezoidal result and $\xi \in [a, b]$.

The trapezoidal rule is easy to implement numerically through the following simple algorithm

- Choose the number of mesh points and fix the step length.
- calculate $f(a)$ and $f(b)$ and multiply with $h/2$.
- Perform a loop over $n = 1$ to $n - 1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $f(a+h) + f(a+2h) + f(a+3h) + \cdots + f(b-h)$. Each step in the loop corresponds to a given value $a + nh$.
- Multiply the final result by h and add $hf(a)/2$ and $hf(b)/2$.

Python offers an extremely versatile programming environment, allowing for the inclusion of analytical studies in a numerical program. Here we show an example code with the **trapezoidal rule** using **SymPy** to evaluate an integral and compute the absolute error with respect to the numerically evaluated one of the integral $4\int_0^1 dx/(1+x^2) = \pi$:

```
from math import *
from sympy import *
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s

#  function to compute pi
def function(x):
    return 4.0/(1+x*x)

a = 0.0;  b = 1.0; n = 100
result = Trapez(a,b,function,n)
print "Trapezoidal rule=", result
# define x as a symbol to be used by sympy
x = Symbol('x')
exact = integrate(function(x), (x, 0.0, 1.0))
print "Sympy integration=", exact
# Find relative error
print "Relative error", abs((exact-result)/exact)
```

The following extended version of the trapezoidal rule allows you to plot the relative error by comparing with the exact result. By increasing to 10^8 points one arrives at a region where numerical errors start to accumulate.

```
from math import log10
import numpy as np
from sympy import Symbol, integrate
import matplotlib.pyplot as plt
# function for the trapezoidal rule
def Trapez(a,b,f,n):
   h = (b-a)/float(n)
   s = 0
   x = a
   for i in range(1,n,1):
       x = x+h
       s = s+ f(x)
   s = 0.5*(f(a)+f(b)) +s
   return h*s
#  function to compute pi
def function(x):
    return 4.0/(1+x*x)
# define integration limits
a = 0.0;  b = 1.0;
# find result from sympy
# define x as a symbol to be used by sympy
x = Symbol('x')
exact = integrate(function(x), (x, a, b))
```

```
# set up the arrays for plotting the relative error
n = np.zeros(9); y = np.zeros(9);
# find the relative error as function of integration points
for i in range(1, 8, 1):
    npts = 10**i
    result = Trapez(a,b,function,npts)
    RelativeError = abs((exact-result)/exact)
    n[i] = log10(npts); y[i] = log10(RelativeError);
plt.plot(n,y, 'ro')
plt.xlabel('n')
plt.ylabel('Relative error')
plt.show()
```

The last example shows the potential of combining numerical algorithms with symbolic calculations, allowing us thereby to

- Validate and verify our algorithms.
- Including concepts like unit testing, one has the possibility to test and validate several or all parts of the code.
- Validation and verification are then included *naturally*.
- The above example allows you to test the mathematical error of the algorithm for the trapezoidal rule by changing the number of integration points. You get trained from day one to think error analysis.

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and height given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}), \tag{40}$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. We will discuss its truncation error below.

The correct mathematical expression for the local error for the rectangular rule $R_i(h)$ for element i is

$$\int_{-h}^{h} f(x)dx - R_i(h) = -\frac{h^3}{24}f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - R_h(f) = -\frac{b-a}{24}h^2 f^{(2)}(\xi),$$

where R_h is the result obtained with rectangular rule and $\xi \in [a,b]$.

Instead of using the above first-order polynomials approximations for f, we attempt at using a second-order polynomials. In this case we need three points in order to define a second-order polynomial approximation

$$f(x) \approx P_2(x) = a_0 + a_1 x + a_2 x^2.$$

Using again Lagrange's interpolation formula we have

$$P_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0.$$

Inserting this formula in the integral of Eq. (37) we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}\left(f_h + 4f_0 + f_{-h}\right) + O(h^5),$$

which is Simpson's rule.

Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is again the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (36) to be

$$I = \int_a^b f(x)dx = \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b\right), \tag{41}$$

with a global error which goes like $O(h^4)$.

More formal expressions for the local and global errors are for the local error

$$\int_a^b f(x)dx - \frac{b-a}{6}\left[f(a) + 4f((a+b)/2) + f(b)\right] = -\frac{h^5}{90}f^{(4)}(\xi),$$

and for the global error

$$\int_a^b f(x)dx - S_h(f) = -\frac{b-a}{180}h^4 f^{(4)}(\xi).$$

with $\xi \in [a,b]$ and S_h the results obtained with Simpson's method.

Algorithm for Simpson's rule

The method can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots + 4f(b-h)$. Each step in the loop corresponds to a given value $a + nh$. Odd values of n give 4 as factor while even values yield 2 as factor.
- Multiply the final result by $\frac{h}{3}$.

Summary for equal-step methods

In more general terms, what we have done here is to approximate a given function $f(x)$ with a polynomial of a certain degree. One can show that given $n+1$ distinct points $x_0, \ldots, x_n \in [a,b]$ and $n+1$ values $y_0, \ldots, y_n$ there exists a unique polynomial $P_n(x)$ with the property

$$P_n(x_j) = y_j \qquad j = 0, \ldots, n$$

In the Lagrange representation the interpolating polynomial is given by

$$P_n = \sum_{k=0}^{n} l_k y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} \quad k = 0, \ldots, n.$$

If we for example set $n = 1$, we obtain

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} x - \frac{y_1 x_0 + y_0 x_1}{x_1 - x_0},$$

which we recognize as the equation for a straight line.

The polynomial interpolatory quadrature of order n with equidistant quadrature points $x_k = a + kh$ and step $h = (b-a)/n$ is called the Newton-Cotes quadrature formula of order n.