PHY 312

Numerical Methods Project

Perihelion precession of Mercury

Group Members:

Haris Avudaiyappan M(16027)

Adarsh Kumar Dash (17003)

Kiran Rewatkar(17054)

Amrapali Sen(17010)

Gulshan Khurana (17042)

Introduction

Here we will introduce Newtonian dynamics , concept of differential equations, their discretization based on Newton's law of gravitation and its possible extensions. Further we will visualize the trajectories using VPython. We will also develop tools to extract the relevant quantity from the result of the simulation and finally present the principle of dimensional analysis to estimate the size of the effect studied as well as its
expected accuracy.

We say we understand a physical system if the assumed forces acting on it lead to the observed trajectories. Here it is like an initial value problem where we need to calculate the location in space of the object of interest at any future point in time, once the initial conditions are fixed properly. We will neglect the finite size of the object and its orientation in space and parametrise location by a single three vector $\boldsymbol{r}(t)$. We will also be able to calculate the object's velocity $\boldsymbol{v}(t)$ and acceleration $\boldsymbol{a}(t)$ in order to describe and control its dynamics. The velocity describes a change of location over time, for some infinitesimally small t, can be written as:

$$\boldsymbol{r}(t + \Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\,\Delta t + \ldots\ldots \qquad (1)$$

Similarly we can describe acceleration as:

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) + \boldsymbol{a}(t)\,\Delta t + \ldots\ldots \qquad (2)$$

In writing both (1) and (2) we have safely ignored the higher values of Δt , considering it to be very small.

Also , we define the time derivative as:

$$\boldsymbol{v}(t) = \lim_{\Delta t \to 0} \Delta\boldsymbol{r}/\Delta t \simeq d\boldsymbol{r}/dt = \boldsymbol{r}(t) \qquad (3)$$

where $\Delta\mathbf{r}(t) = \boldsymbol{r}(t + \Delta t) - \boldsymbol{r}(t)$.
Similarly we can also write:

$$\boldsymbol{a}(t) = \lim_{\Delta t \to 0} \Delta\boldsymbol{v}/\Delta t \simeq d\boldsymbol{v}/dt = \dot{\boldsymbol{v}}(t)$$
$$= d^2 r/\, dt^2 = \ddot{\boldsymbol{r}}(t)$$

Newton observed that, if a body is at rest, it will remain at rest, and if it is in motion it will remain in motion at a constant velocity in a straight line, unless it is acted upon by some external force and this law is known as Newton's first law. Through Newton's second law of motion we calculate the force $\boldsymbol{F}$ by changing the motion of some object quantified as:

$$\boldsymbol{F}(\mathbf{r},t)= d(m\mathbf{v})/dt \qquad (4)$$

Considering that the mass does not change with time the second law of motion reduces:

$$\begin{aligned}\boldsymbol{F}(\mathbf{r}) &= m\,\boldsymbol{a}(t)\\ &= m\,\dot{\boldsymbol{v}}(t)\\ &= m\,\ddot{\boldsymbol{r}}(t)\end{aligned} \qquad (5)$$

In general force can also depend on the time or the velocity of the object but here, we restrict ourselves to the case where the force depends on only the location . Hence as soon as the force $\boldsymbol{F}(\mathbf{r})$ is known for all $\mathbf{r}$ we can in principle calculate the trajectory by solving Eq. (5) for $\boldsymbol{r}(t)$.

General Relativity and The Perihelion Motion of Mercury

Newton's law of gravitation is ,

$$\vec{F}_N(\vec{r}) = -\frac{G_N m M_\odot}{r^2}\frac{\vec{r}}{r} \qquad (6)$$

Where G_N is a universal gravitational constant, m is mass of mercury, $M_\odot$ is mass of the Sun and r is the distance between the Sun and the Mercury. We assume that the Sun is the coordinate system and much much larger than Mercury. Schwarzschild radius of Sun is introduced for later convenience:

$$r_S = \frac{2G_N M_\odot}{c^2} = 2.95\ \text{km} \qquad (7)$$

c is the speed of light. Another quantity of dimensions G_N, $M_\odot$ and c cannot be formed, hence r_S is the characteristic length scale of the gravitational field of the Sun. Now Newton's second law can be written as,

$$\ddot{\vec{r}} = -\frac{c^2}{2}\left(\frac{r_S}{r^2}\right)\frac{\vec{r}}{r}\,. \qquad (8)$$

Orbits with potential $1/r$, have an elliptical path and are fixed in space, and the perihelion, i.e. , the closest approach point to the sun does not move. But when potential deviates from $1/r$ and vanishes for $r\to\infty$, perihelion does move. Hence we can say perihelion is very sensitive to gravitational potential .

Current motion of perihelion of mercury is

$$(574.10 \pm 0.65)'' \text{ per 100 earth years.}$$

$''$ denotes arc seconds : $1''= (1/3600)°$. Other planet's gravitational force also acts on Mercury, by staying within newtonian theory, the bulk of this number can be explained, but there was a unexplained residual motion of

$$\delta\Theta_M = (42.56 \pm 0.94)'' \text{ per 100 earth years}$$

until Einstein quantified the prediction of general relativity,

$$\delta\Theta_{GR} = (43.03 \pm 0.03)'' \text{ per 100 earth years.}$$

For perihelion to move, we need some modification in force, such that it depends on r and vanishes at $r \to \infty$. Based on this discussion, we multiply the right hand side of equation (8) with $(1+ \alpha * r_S/r)$, where α is some dimensionless parameter. This adds the smallest possible deviation from a $1/r$ potential, which is expressed relative to the characteristic length r_S. r_S was for the gravitational potential of the Sun, but there is also a characteristic parameter r_L for the dynamics of Mercury,

$$r_L^2 := \frac{\vec{L}^2}{m^2 c^2} = \frac{(\vec{r} \times \dot{\vec{r}})^2}{c^2} \ , \tag{9}$$

$\boldsymbol{L}$ Angular momentum, also it is a constant of motion for central potential. Analogous to the above discussion we add a parameter of ($\beta * r_L^2/r^2$) to this equation. Here we chose r_L^2/r^2 and not r_L/r because the correlation term must be a scalar quantity, and vectors can be written as scalar product, also we cannot use square root of scalar product because wrong mathematical properties will impose the equations of motion. Radius is the only allowed vector to appear as its length in linear order, since radius is related to geometrical properties of the system. By combining all things with get ,

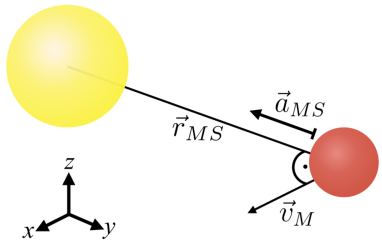
$$\ddot{\vec{r}} = -\frac{c^2}{2}\frac{r_S}{r^2}\left(1 + \alpha\frac{r_S}{r} + \beta\frac{r_L^2}{r^2}\right)\frac{\vec{r}}{r} \ . \tag{10}$$

From the parameters of Mercury at its perihelion r_L^2 can be estimated: corresponding velocity is $\dot{r}(t=0) = |\boldsymbol{v}_M(0)| = 59.0$ km/s and the closest distance between Mercury and sun is $r_{MS} = |\boldsymbol{r}_{MS}(0)| = 46.0 * 10^6$ km. $\boldsymbol{r}$ and $\dot{\boldsymbol{r}}$ are perpendicular to each other. Thus we get $(r_L^2/r_{MS}^2) \sim 4 * 10^{-8}$ km and $(r_S/r_{MS}) \sim 6 * 10^{-8}$. We can find the parameters α and β from fit to data. The parameters can be calculated from the theory used. Special theory gives the actual values.

From general relativity we get $\alpha = 0$ and $\beta = 3$. We use these values and simulate to calculate the perihelion motion of Mercury.

Numerical Implementation

Describing the motion and visualisation of orbit



We start the simulation with the "initial" distance and velocity as given in equations (1) and (2). We are assuming here that the Sun has an infinite mass and hence remains fixed in the coordinate system.

We start with values of perihelion of Mercury $|\mathbf{r}_{MS}(0)| = 46.0 \cdot 10^6$ km and $|\mathbf{v}_M(0)| = 59.0$ km/s as initial (for t = 0) parameters.

As the computer does not understand the physical units, we replace it with parameters in a natural range, i.e., by expressing distances in $R_0 = 10^7$ km $= 10^{10}$ m and time intervals in $T_0 = 1$ earth day. One Mercury year = $T_M = 88.0\ T_0$.

Then the initial distance for the simulation to start and the initial velocity with the acceleration prefactor in equation (10) becomes

$$r_{MS}(0) = 4.60 R_0\,, \quad v_M(0) = 0.510 \frac{R_0}{T_0}$$

$$a_{MS}(r_{MS}) = \frac{c^2}{2}\frac{r_S}{r_{MS}^2} = 0.990 \frac{R_0}{T_0^2}\frac{1}{(r_{MS}/R_0)^2}$$

In the code it reads,

```
rM0 = 4.60
vM0 = 5.10e-1
c_a = 9.90e-1
TM = 8.80e+1
rS = 2.95e-7
rL2 = 8.19e-7
```

Here, we will be working on VPython module to visualize the orbits for the implemented codes and make observations. We take here the Sun as the centre of the coordinate system (and at rest) and the velocity of Mercury being perpendicular to the position vector of Mercury from the Sun to Mercury. The orbit is visualised using *M.trajectory* in white color.

```
M = sphere(pos=vector(0, rM0, 0), radius=0.5, color=color.red)
S = sphere(pos=vector(0, 0, 0), radius=1.5, color=color.yellow)
M.velocity = vector(vM0, 0, 0)
S.velocity = vector(0, 0, 0)
```

```
M.trajectory = curve(color=color.white)
```

The function *evolve_mercury* takes the input of previous location and velocity of Mercury with the parameters α and β and evolves the position and velocity of the planet in accordance with the equation (10). The factor $\frac{1}{Nt}$ in the *dt* expression gives the steps with which we calculate location and velocity of the planet at t_0 + dt using *evolve_mercury* as given below.

```
def evolve_mercury(vec_rM_old, vec_vM_old, alpha, beta, Nt):
 # Compute the strength of the acceleration
    dt = 2 * vM0 / c_a / Nt
    temp = 1 + alpha * rS / vec_rM_old.mag + beta * rL2 / vec_rM_old.mag**2
    aMS = c_a * temp / vec_rM_old.mag**2
 # Multiply by the direction
    vec_aMS = - aMS * ( vec_rM_old / vec_rM_old.mag )
 # Update velocity vector
    vec_vM_new = vec_vM_old + vec_aMS * dt
# Update position vector
    vec_rM_new = vec_rM_old + vec_vM_new * dt
    return vec_rM_new, vec_vM_new
```

Finally for the visualisation purpose, we use *simulate_mercury* to see the orbital motion of Mercury. Here α and β can be kept as per the choice. Keeping α and β as zero, we may expect the non-realistic case where the precession is not occurring. To get more than one revolutions, we must multiply a factor to *TM* in the *while loop argument*.

```
def simulate_mercury(t, alpha, beta, Nt):
    dt = 2 * vM0 / c_a / Nt
    # Execute the loop as long as t < 2*TM
    while t < 10*TM:
    # Set the frame rate (you can choose a higher rate to accelerate the program)
        rate(10)
    # Update the drawn trajectory with the current position
        M.trajectory.append(pos=M.pos)
    # Update velocity and position
        M.pos, M.velocity = evolve_mercury(M.pos , M.velocity , alpha, beta, Nt)
    # Advance time by one step
        t = t + dt
```

Finding the perihelion shift

Perihelion is defined as the shortest distance between the planet and the sun (around which it is revolving). Taking this into account, we can calculate the perihelion shift if we measure the angle changed between the previous perihelion and the current perihelion (in terms of the position vector).

Now perihelion can be calculated simply as the magnitude of the position vector at time t_p which has magnitude smaller than that at time (t_p - dt) and also smaller than that at time (t_p + dt). In python, it becomes:

```
def Find_perihelion(turns, max_turns, alpha, beta, Nt):

    # Set up vectors
    vec_rM0 = vector(0, rM0, 0)
    vec_rM_last = vec_rM0

    list_perih = list()
    # Find perihelion for each turn and print it out
    while turns < max_turns:
        vec_rM_before_last = vec_rM_last
        # Store position of Mercury in a new vector (since we will change M.pos)
        vec_rM_last = vector(M.pos)
        #<...update Mercury position...>
        # Update velocity and position
        M.pos, M.velocity = evolve_mercury(M.pos , M.velocity , alpha, beta, Nt)
        # Check if at perihelion
        if (vec_rM_last.mag < M.pos.mag) and (vec_rM_last.mag < vec_rM_before_last.mag):
            list_perih.append(vec_rM_last)
            #print(vec_rM_last.mag)
            turns = turns + 1
    return list_perih
```

Then we measure the angle between successive perihelions using

$$\sphericalangle(\vec{v}_1, \vec{v}_2) = \arccos\left(\frac{\vec{v}_1 \cdot \vec{v}_2}{|\vec{v}_1|\,|\vec{v}_2|}\right)$$

And then find the average over an optimum number of turns to minimize error.

```
# Define function for angle extraction
def angle_between(v1, v2):
    return acos( dot(v1, v2) / (v1.mag * v2.mag) ) * 180. / pi
```

```
def perihelion_motion(list_perih, max_turns):
    sum_angle=0.
    for n in range(1, max_turns):
        # Calculate angle
        sum_angle = sum_angle + angle_between(list_perih[n-1], list_perih[n])
    # Display the average
    avg = sum_angle/(max_turns-1)
    return avg
```

The dependence of δΘ on α and β can be calculated individually, while keeping the other constant. We would expect the dependence to be similar as per the discussion of equation (10).

Interpolation using Newton Divided Differences formula

The NDD method is used to interpolate the value of δΘ(α, β) at any given value of α and β.

```
# Functions using the NDD method for interpolation
def proterm(i, value, x):
    pro = 1;
    for j in range(i):
        pro = pro * (value - x[j]);
    return pro;
```

```
def dividedDiffTable(x, y, n):

    for i in range(1, n):
        for j in range(n - i):
            y[j][i] = ((y[j][i - 1] - y[j + 1][i - 1]) /
                                    (x[j] - x[i + j]));
    return y;
```

```
def applyFormula(value, x, y, n):

    sum = y[0][0];

    for i in range(1, n):
        sum = sum + (proterm(i, value, x) * y[0][i]);

    return sum;
```

```
def printDiffTable(y, n):

    for i in range(n):
        for j in range(n - i):
            print(round(y[i][j], 4), "\t",
                                end = " ");

        print("");
```

The variable *alp=0* is used interchangeably for α and β to calculate the dependence of δΘ on the other.

```
# Main Code

# number of inputs given

n = 11;

d1 = [[0 for i in range(20)]
         for j in range(20)];

d2 = [[0 for i in range(20)]
         for j in range(20)];

t = [0 for i in range(n)];
a = [0 for i in range(n)];
b = [0 for i in range(n)];

for i in range(n):
    t[i] = 1.0e04*i;

    # y[][] is used for divided difference
# table where y[][0] is used for input
mercury_turns = 0;
mercury_maxturns = 20;
alp = 0;
N = 200;
```

```
for i in range(n):
    perihm = Find_perihelion(mercury_turns, mercury_maxturns, t[i], alp, N);
    d1[i][0] = perihelion_motion(perihm, mercury_maxturns);
    a[i] = perihelion_motion(perihm, mercury_maxturns);

for i in range(n):
    perihm = Find_perihelion(mercury_turns, mercury_maxturns, alp, t[i], N);
    d2[i][0] = perihelion_motion(perihm, mercury_maxturns);
    b[i] = perihelion_motion(perihm, mercury_maxturns);
```

```
    # calculating divided difference table
y1=dividedDiffTable(t, d1, n);
y2=dividedDiffTable(t, d2, n);

    # value to be interpolated
value = 3.5e04;

# printing the value
print("\nValue at", value, "is",
    round(applyFormula(value, t, y1, n), 2))

# displaying divided difference table
#printDiffTable(y1, n);
plt.title("Fitting using NDD")
plt.grid(True)
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\delta \theta (\alpha, \beta)^{\circ}$')
plt.plot(t,a, color = 'green')
plt.scatter(t,a,color='blue')
plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
plt.show()
```

```
# displaying divided difference table
#printDiffTable(y2, n);
plt.title("Fitting using NDD")
plt.grid(True)
plt.xlabel(r'$\beta$')
plt.ylabel(r'$\delta \theta (\alpha, \beta)^{\circ}$')
plt.plot(t,b, color = 'green')
plt.scatter(t,b,color='blue')
plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
plt.show()
```

The following is the complete program for the implementation required -

[1]:
```
from vpython import vector, sphere, color, curve, rate, acos, pi, dot
import matplotlib.pyplot as plt
```

[2]:
```
rM0 = 4.60
vM0 = 5.10e-1
c_a = 9.90e-1
TM = 8.80e+1
rS = 2.95e-7
rL2 = 8.19e-7
```

[3]:
```
M = sphere(pos=vector(0, rM0, 0), radius=0.5, color=color.red)
S = sphere(pos=vector(0, 0, 0), radius=1.5, color=color.yellow)
M.velocity = vector(vM0, 0, 0)
S.velocity = vector(0, 0, 0)
```

[4]:
```
M.trajectory = curve(color=color.white)
```

[5]:
```
def evolve_mercury(vec_rM_old, vec_vM_old, alpha, beta, Nt):
 # Compute the strength of the acceleration
    dt = 2 * vM0 / c_a / Nt
    temp = 1 + alpha * rS / vec_rM_old.mag + beta * rL2 / vec_rM_old.mag**2
    aMS = c_a * temp / vec_rM_old.mag**2
 # Multiply by the direction
    vec_aMS = - aMS * ( vec_rM_old / vec_rM_old.mag )
 # Update velocity vector
    vec_vM_new = vec_vM_old + vec_aMS * dt
# Update position vector
    vec_rM_new = vec_rM_old + vec_vM_new * dt
    return vec_rM_new, vec_vM_new
```

[6]:
```
def simulate_mercury(t, alpha, beta, Nt):
    dt = 2 * vM0 / c_a / Nt
    # Execute the loop as long as t < 2*TM
    while t < 10*TM:
    # Set the frame rate (you can choose a higher rate to accelerate the program)
        rate(10)
    # Update the drawn trajectory with the current position
        M.trajectory.append(pos=M.pos)
    # Update velocity and position
        M.pos, M.velocity = evolve_mercury(M.pos , M.velocity , alpha, beta, Nt)
    # Advance time by one step
        t = t + dt
```

[7]:
```
def Find_perihelion(turns, max_turns, alpha, beta, Nt):

    # Set up vectors
    vec_rM0 = vector(0, rM0, 0)
    vec_rM_last = vec_rM0

    list_perih = list()
    # Find perihelion for each turn and print it out
    while turns < max_turns:
        vec_rM_before_last = vec_rM_last
        # Store position of Mercury in a new vector (since we will change M.pos)
        vec_rM_last = vector(M.pos)
        #<...update Mercury position...>
        # Update velocity and position
        M.pos, M.velocity = evolve_mercury(M.pos , M.velocity , alpha, beta, Nt)
        # Check if at perihelion
        if (vec_rM_last.mag < M.pos.mag) and (vec_rM_last.mag < vec_rM_before_last.mag):
            list_perih.append(vec_rM_last)
            #print(vec_rM_last.mag)
            turns = turns + 1
    return list_perih
```

[8]:
```
# Define function for angle extraction
def angle_between(v1, v2):
    return acos( dot(v1, v2) / (v1.mag * v2.mag) ) * 180. / pi
```

[9]:
```
def perihelion_motion(list_perih, max_turns):
    sum_angle=0.
    for n in range(1, max_turns):
        # Calculate angle
        sum_angle = sum_angle + angle_between(list_perih[n-1], list_perih[n])
    # Display the average
    avg = sum_angle/(max_turns-1)
    return avg
```

[10]:
```
# Functions using the NDD method for interpolation
def proterm(i, value, x):
    pro = 1;
    for j in range(i):
        pro = pro * (value - x[j]);
    return pro;
```

[11]:
```
def dividedDiffTable(x, y, n):

    for i in range(1, n):
        for j in range(n - i):
            y[j][i] = ((y[j][i - 1] - y[j + 1][i - 1]) /
                                      (x[j] - x[i + j]));
    return y;
```

[12]:
```
def applyFormula(value, x, y, n):

    sum = y[0][0];

    for i in range(1, n):
        sum = sum + (proterm(i, value, x) * y[0][i]);

    return sum;
```

[13]:
```
def printDiffTable(y, n):

    for i in range(n):
        for j in range(n - i):
            print(round(y[i][j], 4), "\t",
                                end = " ");

        print("");
```

[14]:
```
# Main Code

# number of inputs given

n = 11;

d1 = [[0 for i in range(20)]
         for j in range(20)];

d2 = [[0 for i in range(20)]
         for j in range(20)];

t = [0 for i in range(n)];
a = [0 for i in range(n)];
b = [0 for i in range(n)];

for i in range(n):
    t[i] = 1.0e04*i;

    # y[][] is used for divided difference
# table where y[][0] is used for input
mercury_turns = 0;
mercury_maxturns = 20;
alp = 0;
N = 200;
```

```
for i in range(n):
    perihm = Find_perihelion(mercury_turns, mercury_maxturns, t[i], alp, N);
    d1[i][0] = perihelion_motion(perihm, mercury_maxturns);
    a[i] = perihelion_motion(perihm, mercury_maxturns);

for i in range(n):
    perihm = Find_perihelion(mercury_turns, mercury_maxturns, alp, t[i], N);
    d2[i][0] = perihelion_motion(perihm, mercury_maxturns);
    b[i] = perihelion_motion(perihm, mercury_maxturns);
```

```
    # calculating divided difference table
y1=dividedDiffTable(t, d1, n);
y2=dividedDiffTable(t, d2, n);

    # value to be interpolated
value = 3.5e04;

# printing the value
print("\nValue at", value, "is",
    round(applyFormula(value, t, y1, n), 2))

# displaying divided difference table
#printDiffTable(y1, n);
plt.title("Fitting using NDD")
plt.grid(True)
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\delta \theta (\alpha, \beta)^{\circ}$')
plt.plot(t,a, color = 'green')
plt.scatter(t,a,color='blue')
plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
plt.show()
```

```
# displaying divided difference table
#printDiffTable(y2, n);
plt.title("Fitting using NDD")
plt.grid(True)
plt.xlabel(r'$\beta$')
plt.ylabel(r'$\delta \theta (\alpha, \beta)^{\circ}$')
plt.plot(t,b, color = 'green')
plt.scatter(t,b,color='blue')
plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
plt.show()
```

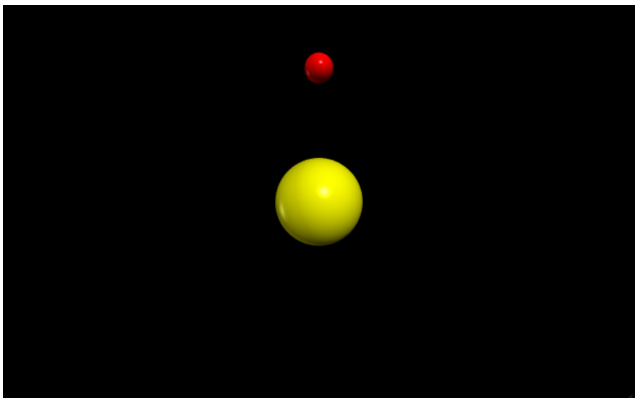
[15]:
```
#simulate_mercury(time of start, alpha, beta, Δt0/Δt ) 1st Sc
simulate_mercury(0, 1.0e06, 0, 20)
```

[16]:
```
m = 20;

n_t = [10, 100, 200, 500];
pm1 = [0 for i in range(m)];
pm2 = [0 for i in range(m)];
pm3 = [0 for i in range(m)];
pm4 = [0 for i in range(m)];

maxturns = [0 for i in range(m)];

for i in range(2, m):
    maxturns[i] = i;

ap = 0;
bt = 0;
mt = 0;

for j in range(m):
    pl1 = Find_perihelion(mt, maxturns[j], ap, bt, n_t[0]);
    pm1[j] = perihelion_motion(pl1, maxturns[j]);

for j in range(m):
    pl2 = Find_perihelion(mt, maxturns[j], ap, bt, n_t[1]);
    pm2[j] = perihelion_motion(pl2, maxturns[j]);

for j in range(m):
    pl3 = Find_perihelion(mt, maxturns[j], ap, bt, n_t[2]);
    pm3[j] = perihelion_motion(pl3, maxturns[j]);
```

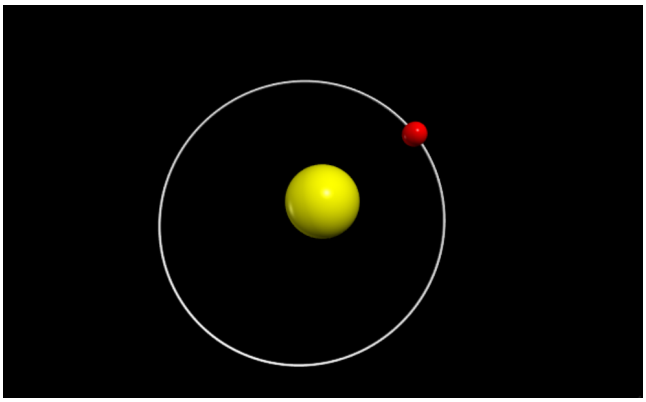
```
for j in range(m):
    pl4 = Find_perihelion(mt, maxturns[j], ap, bt, n_t[3]);
    pm4[j] = perihelion_motion(pl4, maxturns[j]);

plt.title("test")
plt.grid(True)
plt.xlabel('N')
plt.ylabel(r'$\delta \theta (\alpha, \beta)^{\circ}$')
plt.plot(maxturns,pm1, color = 'green')
l1 = plt.scatter(maxturns,pm1,color='blue', label = r'$\Delta t_{0} / \Delta t = 10$')
plt.plot(maxturns,pm2, color = 'green')
l2 = plt.scatter(maxturns,pm2,color='red', label = r'$\Delta t_{0} / \Delta t = 100$')
plt.plot(maxturns,pm3, color = 'green')
l3 = plt.scatter(maxturns,pm3,color='yellow', label = r'$\Delta t_{0} / \Delta t = 200$')
plt.plot(maxturns,pm4, color = 'green')
l4 = plt.scatter(maxturns,pm4,color='black', label = r'$\Delta t_{0} / \Delta t = 500$')
plt.legend(handles=[l1, l2, l3, l4], loc='lower right')
#plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
plt.show()
```

Results

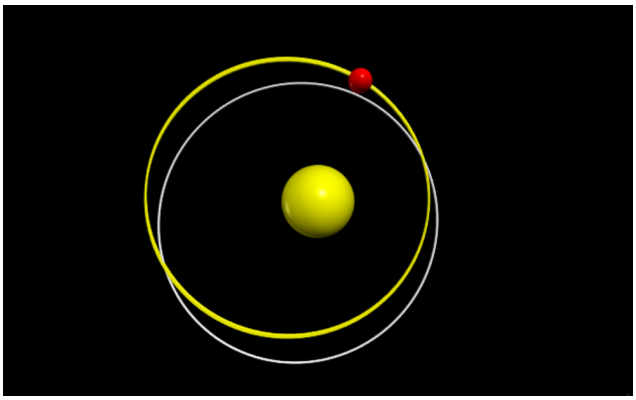
Visualization of the orbit



Case 1 : Initial condition

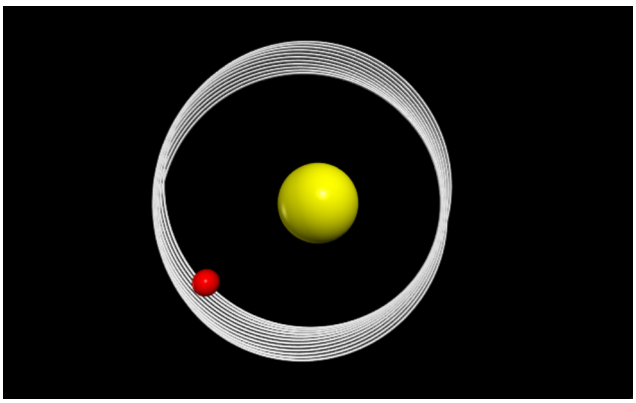


Case 2 : $\alpha = 0$, $\beta = 0$ and $\Delta t = \Delta t_0/20$

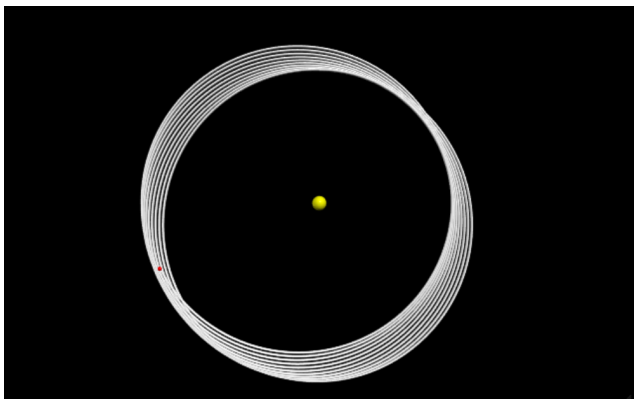


Case 3 : $\alpha = 0$, $\beta = 0$ and $\Delta t = \Delta t_0 \times 1$

Here the white line represents the orbit of 'Case 2' and the yellow line represents the orbit for larger time steps.



Case 4 : $\alpha = 10^6$, $\beta = 0$ and $\Delta t = \Delta t_0 /20$

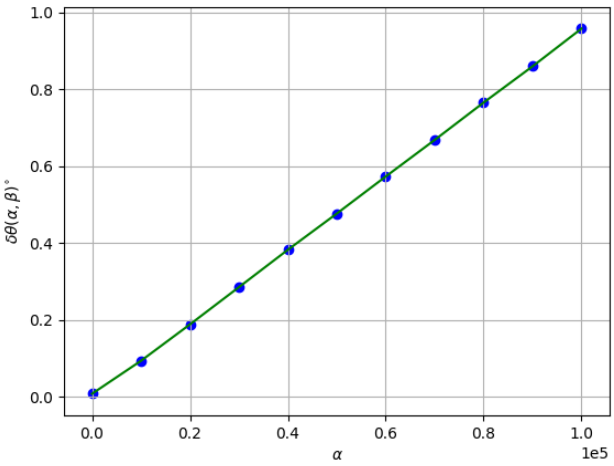


Case 5 : $\alpha = 10^6$, $\beta = 0$ and $\Delta t = \Delta t_0 /20$, but $r_{MS}(0) = 6R_0$

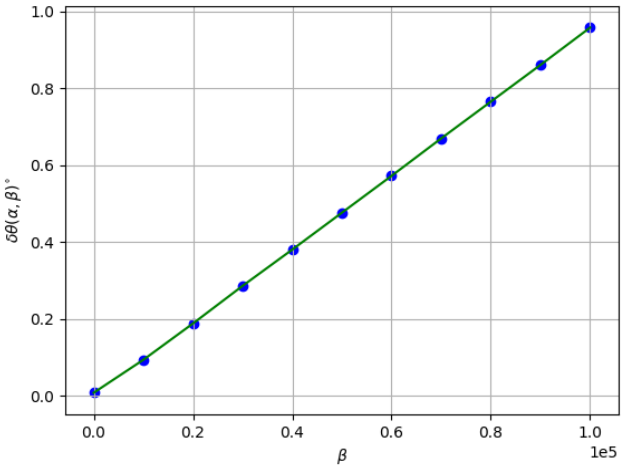
➔ Precession of perihelion was observed for 10 turns in case 4 and case 5 only.

Dependence of perihelion motion (δΘ) on α and β

Linear relation between α and the perihelion motion δΘ for $\Delta t = 2v_M(0)/a_M(0)/200$ and $\beta = 0$



Linear relation β between and the perihelion motion δΘ for $\Delta t = 2v_M(0)/a_M(0)/200$ and $\alpha = 0$



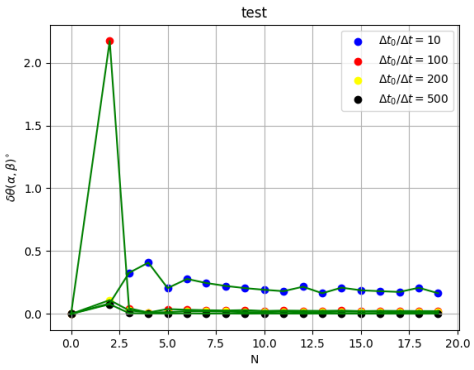
We conclude that the δΘ depends on α or β linearly as : $\delta\Theta(\alpha, \beta) = m_\alpha \cdot \alpha + m_\beta \cdot \beta$

Doing the interpolation to check real conditions, we get $\delta\Theta(\alpha = 0, \beta = 10^5) = 0.96^o$

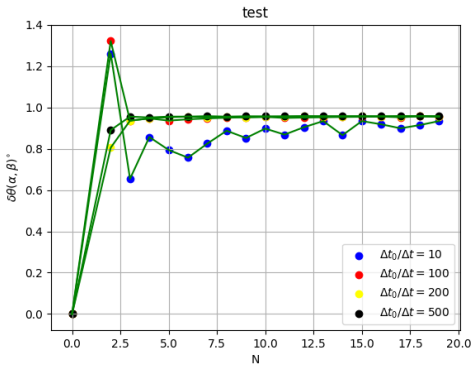
Then $\delta\Theta(\alpha = 0, \beta = 3) = (0.96^o/10^5)*3 = 0.1036''$
One earth year ≈ 4.15 Mercury years. Then the perihelion motion per 100 earth years is $0.1036'' * 415 = 42.99'' \approx 42.3''$.

This agrees greatly with the values observed i.e. $\delta\Theta_M = (42.56 \pm 0.94)''$

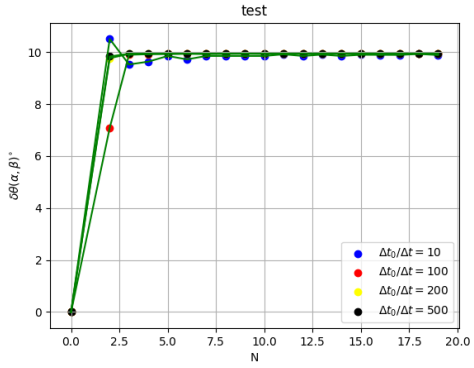
Tests of stability and error analysis



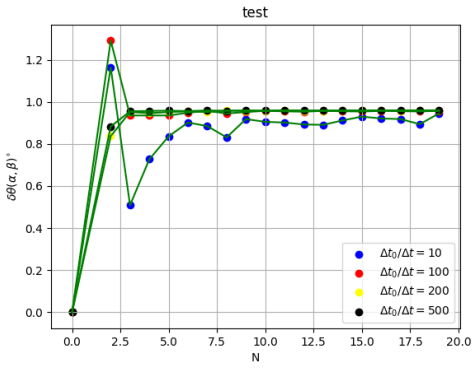
$\alpha = 0, \beta = 0$



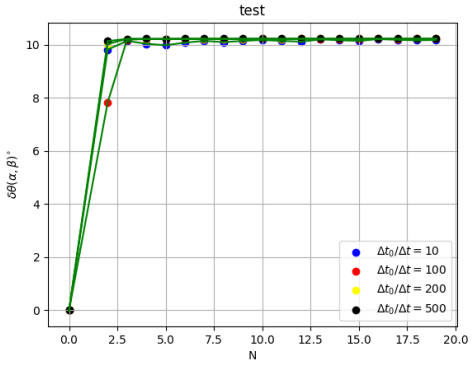
$\alpha = 10^5, \beta = 0$



$\alpha = 10^6, \beta = 0$



$\alpha = 0, \beta = 10^5$



$\alpha = 0, \beta = 10^6$

➔ Perihelion shift (δΘ) was found to be stable for β at orders of 10^6 and to time steps divided by about 500.

Discussion

When we try to have variation with values of α , β and Δt , we notice that large values of α and β give proper noticeable and better rotational trajectories that makes it easier to reduce the systematic errors.

- In Case 2 (Initial case) : $\alpha = 0$, $\beta = 0$ and $\Delta t = \Delta t_0/20$ here Δt is small enough for smooth simulation of rotation of mercury but also simulation takes time accordingly.
- In Case 3 : $\alpha = 0$, $\beta = 0$ and $\Delta t = \Delta t_0$, Δt is much larger than case 2 . In the figure of the result of the simulation we can see that the shift in the perihelion position of mercury as well as change in the orbit of mercury can be noticed easily, hence keeping Δt as small as possible will lead the result precisely.
- When we observe the original motion of mercury, case 2 or case 3 explained above cannot be applied practically. General theory of Relativity plays a major role in the original nature of motion(perihelion) of mercury around the sun, i.e. in Case 4: $\alpha = 10^6$, $\beta = 0$ and $\Delta t = \Delta t_0 /20$ generally for simulation of perihelion motion, value of α or β must be of the order of 10^6 with $\Delta t = \Delta t_0 /20$ for better precession.
- In Case 5: where $r_{MS}(0) = 6R_0$ larger distance between sun and mercury will simulate the perihelion trajectory for N number of turns explicitly.
- With different values of α or β in the above graphs (page 14), showing that the dependence $\boldsymbol{\delta\Theta}$ by keeping one of the GTR coefficient zero. Affect on angle $\boldsymbol{\delta\Theta}$ shows similar variation(i.e. proportional) in angle in degree.

There are many potential sources of uncertainties in a simulation, computational limit and numerical approximations etc. Instead, we focus on the most accessible source: Numerical errors due to finite time steps Δt. While observing for large time-steps (Δt) the values of perihelion motion oscillate (see graph in pg.14) between very discrete values with respect to different number of turns (N) or rotation of mercury around the sun. The scratched code done here is to find the closest distance from the sun for each of the turns. Sometimes the program finds a point before and sometimes after the perihelion causing the oscillations observed. The accuracy of measuring the closest distance of mercury from the sun mostly depends on infinitely small Δt, the bigger value of Δt will generate major errors for the closest distance.

Additional sources of errors include the omission of terms in Equations : $r(t + t) = r(t) + v(t)t$ +..... And $v(t + t) = v(t) + a(t)t$ + ….. And the infinite mass approximation of the Sun. Here we have assumed the position of the sun is fixed for better understanding the situation. The

simulation reproduces the actual trajectory only in the limit $\Delta t \to 0$. Thus, in any numerical simulation programmer/observer always has to identify a proper compromise between numerical accuracy and time spent for the simulation (for $\Delta t \to 0$ the computing time goes to infinity). Thus observer will have to choose a proper values for all parameters Δt, α, β, so that errors can be set minimum as possible and observation period can be feasible. The absolute value of the numerical error mostly depends on the time step Δt and is roughly independent of the values of α and β . Therefore, it is more desirable to pick large values for α and β, because this increases the absolute size of the perihelion motion and thus decreases the relative numerical error.

Nevertheless, the results we got for the perihelion shift were within the error limit given by the observation values. Using the interpolation method, different values of shift can be calculated, keeping in mind that this works good for large values ($\sim 10^5$) only. Thus one will have to measure for a large value and then multiply it with the appropriate conversion factor.

Extending this further, it might be interesting to abandon the simplification of a stationary Sun, as it nicely illustrates Newton's third law. Here it might also be possible to reduce the mass of the Sun to have a more visible result.

{THANK YOU}

References

1. G. M. Clemence. The relativity effect in planetary motions. Rev. Mod. Phys., 19:361–364, Oct 1947.

2. Albert Einstein. Erklärung der Perihelbewegung der Merkur aus der allgemeinen Relativitätstheorie. Sitzungsberichte der königlich preussischen Akademie der Wissenschaften, 18.11.1915.

3. Dr. David R. Williams. Mercury fact sheet. https://nssdc.gsfc.nasa.gov/planetary/factsheet/mercuryfact.html. Accessed: 2017-12-07.

4. James D. Wells. When effective theories predict: The Inevitability of Mercury's anomalous perihelion precession. Masses, 2012.

5. Python Software Foundation. Python. https://www.python.org/. Accessed: 2017-12-07.

6. The Matplotlib development team. Matplotlib. https://matplotlib.org. Accessed: 2017-12-07.

7. David Scherer et al. Vpython - 3d programming for ordinary mortals. http://vpython.org. Accessed: 2017-12-07.

8. Ernst Hairer, Christian Lubich, and Gerhard Wanner. Geometric numerical integration illustrated by the störmer/verlet method. Acta Numerica, 12:399–450, 2003.