UChicago Physics Sample Exercises Documentation

Release 0.1

The University of Chicago

July 22, 2014

CONTENTS

1 `Orbital` – Launching a rocket from Mars to Earth **1**
- 1.1 Build/Run . . . 1
- 1.2 Sample Parameter file . . . 1
- 1.3 Output . . . 2
- 1.4 Plots . . . 3
- 1.5 Methods . . . 3

2 `doublePendulum` – Solving the Classical Double Pendulum **5**
- 2.1 Build/Run . . . 5
- 2.2 Sample Parameter file . . . 5
- 2.3 Output . . . 6
- 2.4 Plots . . . 6
- 2.5 Methods . . . 6

3 `electronTrajectory` – Comparing Trajectory of Charged Particles in a Magnetic Field **11**
- 3.1 Build/Run . . . 11
- 3.2 Output . . . 11
- 3.3 Plots . . . 12
- 3.4 Methods . . . 12

Python Module Index **15**

Index **17**

CHAPTER

ONE

ORBITAL – LAUNCHING A ROCKET FROM MARS TO EARTH

This problem asks the student to launch a rocket from the orbit of Mars on a course for intercept with Earth.

1.1 Build/Run

```
$ python orbital.py
```

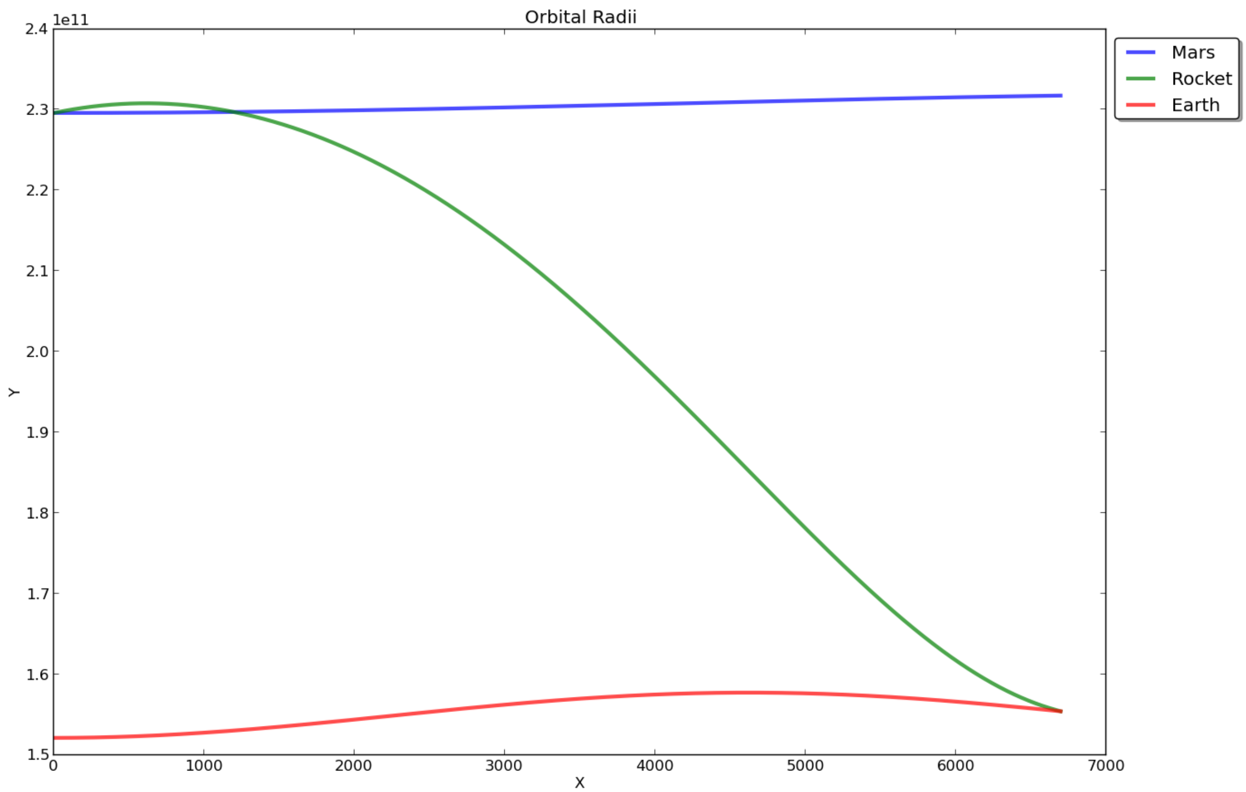
1.2 Sample Parameter file

```
#############################################################
# Constants and initial parameters for Martian invasion
#############################################################

import numpy as np

MAX_T    = 2.43e7
# MAX_T    = 3.0e7
dt       = 3.6e3    # time step of 1 hour
MAX_STEP = int(MAX_T/dt)
G        = 6.67e-11 # Gravitational constant [N m^2/kg^2]

DIAM_E   = 1.273e7  # Diameter of the earth
RADIUS_E = 1.521e11 # Earth to sun           [meters]
RADIUS_M = 2.296e11 # mars to sun            [meters]
RADIUS_R = 9.281e6  # Mars to rocket orbit   [meters]

MASS_S   = 1.989e30 # Mass of the Sun        [kg]
MASS_E   = 5.972e24 # Mass of Earth          [kg]
MASS_M   = 6.417e23 # Mass of Mars           [kg]
MASS_R   = 1.072e6  # Mass of rocket         [kg]

VEL_E    = 2.98e4   # Velocity of Earth      [m/s]
VEL_M    = 2.41e4   # Velocity of Mars       [m/s]
VEL_R    = 2.14e3   # Velocity of rocket     [m/s]

R1 = RADIUS_M-RADIUS_R
```

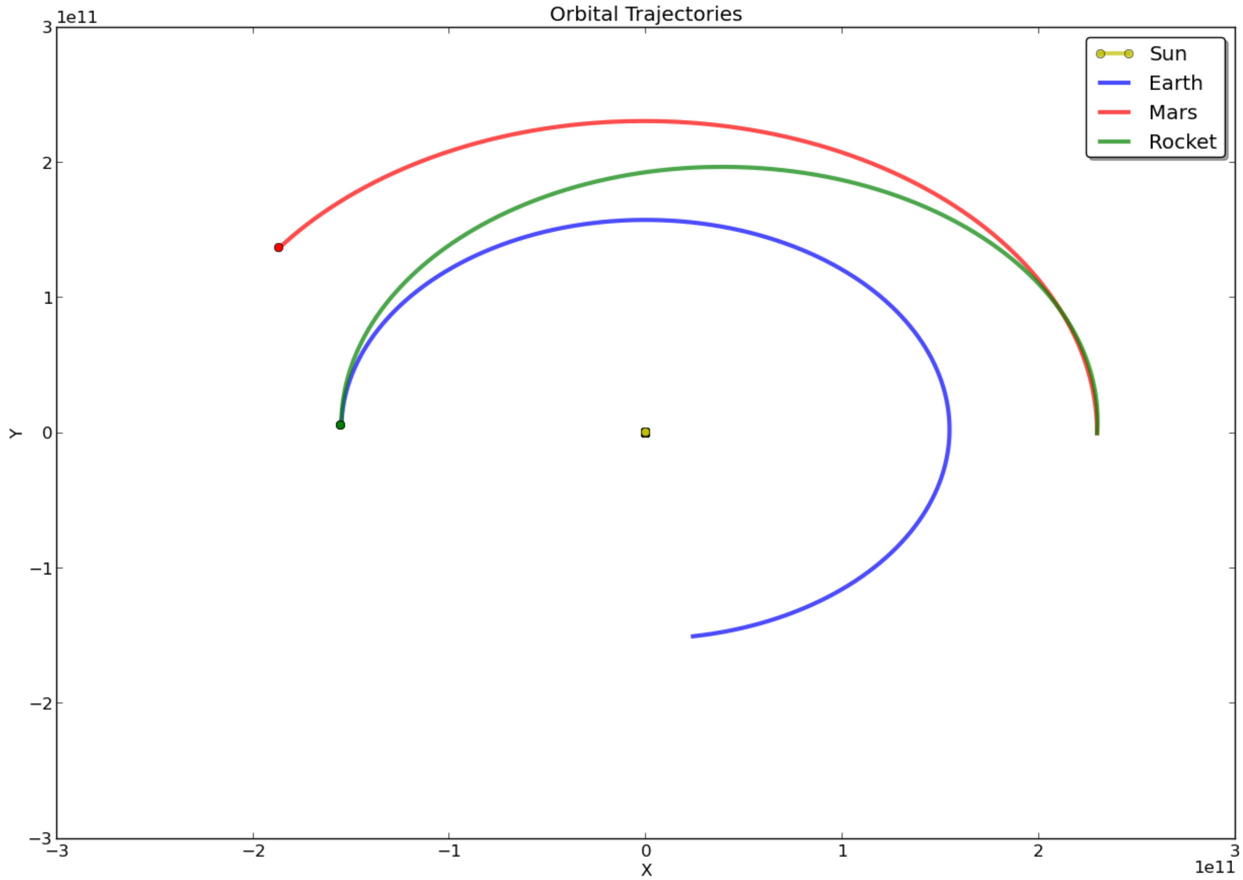
```
# Hohmann boost
V_Y = np.sqrt(G*MASS_S/R1)*(np.sqrt(2*RADIUS_E/(RADIUS_E+R1))-1)

# correction for numerical error of integration techinique
V_Y *= .97

# Secondary Hohmann boost
V_Y2 = np.sqrt(G*MASS_S/RADIUS_E)*(1-np.sqrt(2*R1/(RADIUS_E+R1)))
```

1.3 Output

```
Starting calculation.
Applying primary Hohmann boost.
Calculating trajectory.
LANDED
Plotting.
Saving plot to trajectories.png.
Plotting radii.
Saving plot to radii.png.
```



1.4 Plots



1.5 Methods

orbital.**calculate_forces**(*positions*, *masses*)

Sum the forces on each body in the current system

Parameters

- **positions** – a two dimensional numpy array of floats: each row is a displacement vector belonging to a planetary body
- **masses** – a numpy array of floats: the masses of each body in kg

Returns two dimensional numpy array of floats: an array of 2D force vectors specifying the net force on each body

orbital.**calculate_trajectory**(*V_Y*, *THETA*, *ADJUSTMENT*)

Calculates the trajectory of the rocket given the initial Hohmann velocity boost plus gravity well adjustment

Parameters

- **V_Y** – float: the initial Hohmann velocty boost in m/s
- **THETA** – the initial angular separation of Earth and Mars in radians
- **ADJUSTMENT** – the velocity boost adjustment needed to escape the gravity well

`orbital.`**`force_gravity`**`(`*r1*, *r2*, *m1*, *m2*`)`

Calculates the force of gravity given displacement vectors r1 & r1 and scalar masses m1, m2

Parameters

- **r1** – numpy array of floats: the 2D cartesian location of the first body in meters
- **r2** – numpy array of floats: the 2D cartesian location of the second body in meters
- **m1** – float: the mass of the first body in kg
- **m2** – float: the mass of the first body in kg

Returns float: the gravitational force between the two bodies in Newtons

`orbital.`**`plot_orbit`**`(`*trajectories*`)`

plots the trajectory of each of the bodies from a birds-eye view of the Solar System. Saves output to ./trajectories.png

Parameters **trajectories** – a three dimensional numpy array: first index is time, second index is body (Sun, Earth, etc.), third index is coordinate

Returns `None`

`orbital.`**`plot_radii`**`(`*trajectories*`)`

plots the distance of each body from the Sun over time

Parameters **trajectories** – a three dimensional numpy array: first index is time, second index is body (Sun, Earth, etc.), third index is coordinate

Returns `None`

`orbital.`**`update_pos`**`(`*positions*, *velocities*, *masses*`)`

Update the positions using velocity Verlet integration

Parameters

- **positions** – a two dimensional numpy array of floats: each row is a displacement vector belonging to a planetary body
- **positions** – a two dimensional numpy array of floats: each row is a velocity vector belonging to a planetary body
- **masses** – a numpy array of floats: the masses of each body in kg

Returns tuple length 2 of two dimensional numpy array of floats: the positions and velocities

CHAPTER

TWO

DOUBLEPENDULUM – SOLVING THE CLASSICAL DOUBLE PENDULUM

Solve the classical double pendulum problem. This problem is appropriate for an intermediate mechanics course teach Lagrangian and Hamiltonian Dynamics.

2.1 Build/Run

```
$ python doublePendulum.py
```

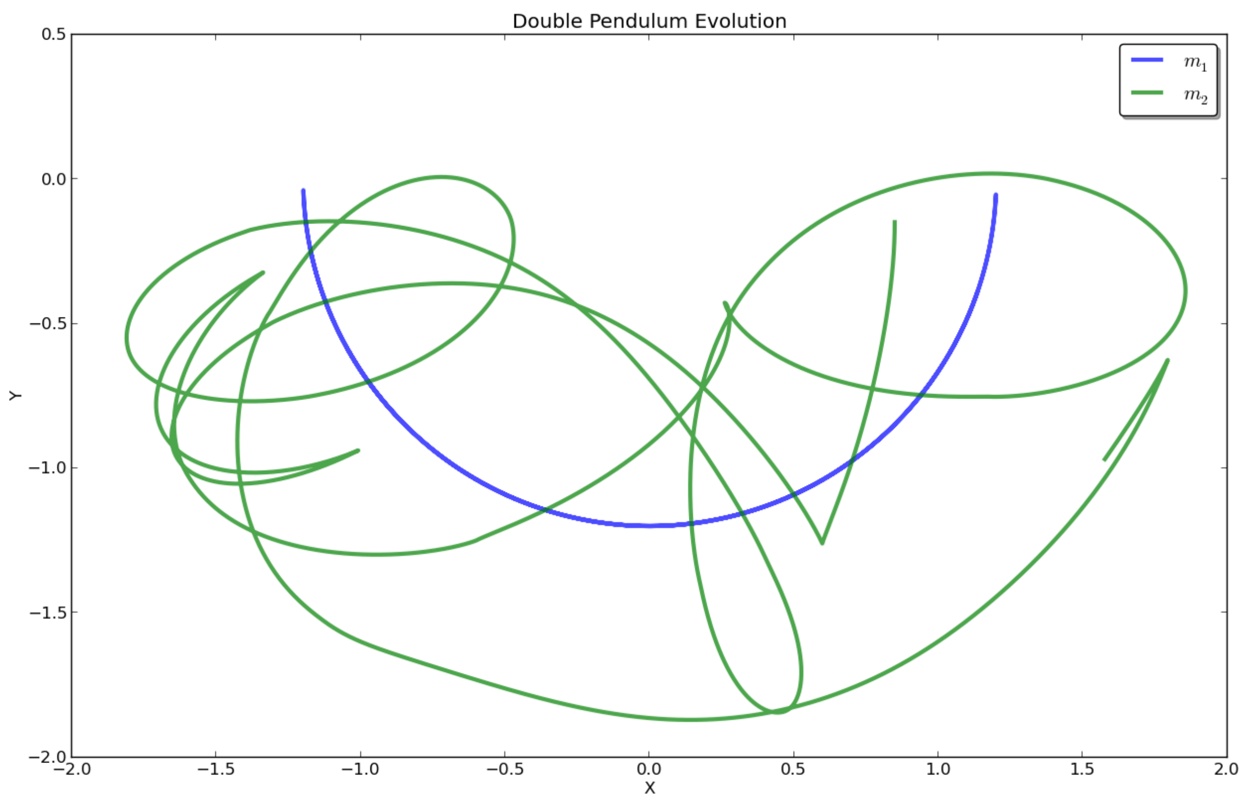
2.2 Sample Parameter file

```
############################################################
# Double Pendulum Parameter file
############################################################

import numpy as np

g = 9.8                    # m/s

l1        = 1.2            # m
l2        = .7             #m

theta1_0 = np.pi/4
theta2_0 = np.pi

m1        = .10            # kg
m2        = .05            # kg

dt        = 1e-3
max_t     = 5.0

nsteps    = int(max_t/dt) # number of steps

print "nsteps:", nsteps
```

2.3 Output

```
nsteps: 5000
Starting calculation.
Plotting.
Saving plot to pendulum.png.
nsteps: 5000
Starting calculation.
Plotting.
Saving plot to pendulum.png.
```

2.4 Plots



2.5 Methods

`doublePendulum.`**`C1`**(*theta1*, *theta2*, *p1*, *p2*)

helper function to calculate a constant `C_1`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, constant used in calculating Hamilton's equation

`doublePendulum.`**`C2`**`(`*theta1*, *theta2*, *p1*, *p2*`)`
helper function to calculate a constant `C_2`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, constant used in calculating Hamilton's equation

`doublePendulum.`**`calculate_paths`**`(`*method='euler'*`)`
use a default or specified method of integration to solve the pendulum's motion

Parameters **method** – optional, string. specify the integration method to use

`doublePendulum.`**`deriv`**`(`*t*, *y*`)`
calculated the derivative of `theta1, theta2, p1, p2`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns numpy array, time derivative of parameters

`doublePendulum.`**`dp1`**`(`*theta1*, *theta2*, *p1*, *p2*, *c1*, *c2*`)`
the time derivative of `p1`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, the time derivative of p1

`doublePendulum.`**`dp2`**`(`*theta1*, *theta2*, *p1*, *p2*, *c1*, *c2*`)`
the time derivative of `p2`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, the time derivative of p2

`doublePendulum.`**`dtheta1`**`(`*theta1*, *theta2*, *p1*, *p2*`)`
the time derivative of `theta1`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, the time derivative of theta1

`doublePendulum.`**`dtheta2`**(*theta1*, *theta2*, *p1*, *p2*)

the time derivative of `theta2`

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns float, the time derivative of theta2

`doublePendulum.`**`euler`**(*theta1*, *theta2*, *p1*, *p2*)

use a naive euler integration schemed to make a single step in the pendulum's motion

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns tuple of updated parameters

`doublePendulum.`**`velocity_verlet`**(*theta1*, *theta2*, *p1*, *p2*)

TODO use a velocity verlet integration schemed to make a single step in the pendulum's motion

Parameters

- **theta1** – float
- **theta2** – float
- **p1** – float
- **p2** – float

Returns tuple of parameters

`plotting.`**`animate`**(*i*)

Helper function for `animate_paths()`. Adds the objects and paths to the plot for each frame

Parameters **i** – the frame number

Returns `None`

`plotting.`**`animate_paths`**(*paths*, *dt*)

Animate the images by compiling .png outputs into a .mp4 video. **REQUIRES: ffmpeg**

Parameters

- **paths** – a three dimensional numpy array: The first index is over time, the second specifies which mass, the third specifies the cartesian displacement
- **dt** – the timestep in seconds

Returns `None`

`plotting.`**`plot_paths`**`(`*paths*`)`

Plot the paths to a png image `./pendulum.png`.

Parameters paths – a three dimensional numpy array: The first index is over time, the second specifies which mass, the third specifies the cartesian displacement

Returns `None`

CHAPTER

THREE

ELECTRONTRAJECTORY – COMPARING TRAJECTORY OF CHARGED PARTICLES IN A MAGNETIC FIELD

Calculate the trajectory of charged particles with different masses in an external magnetic field.

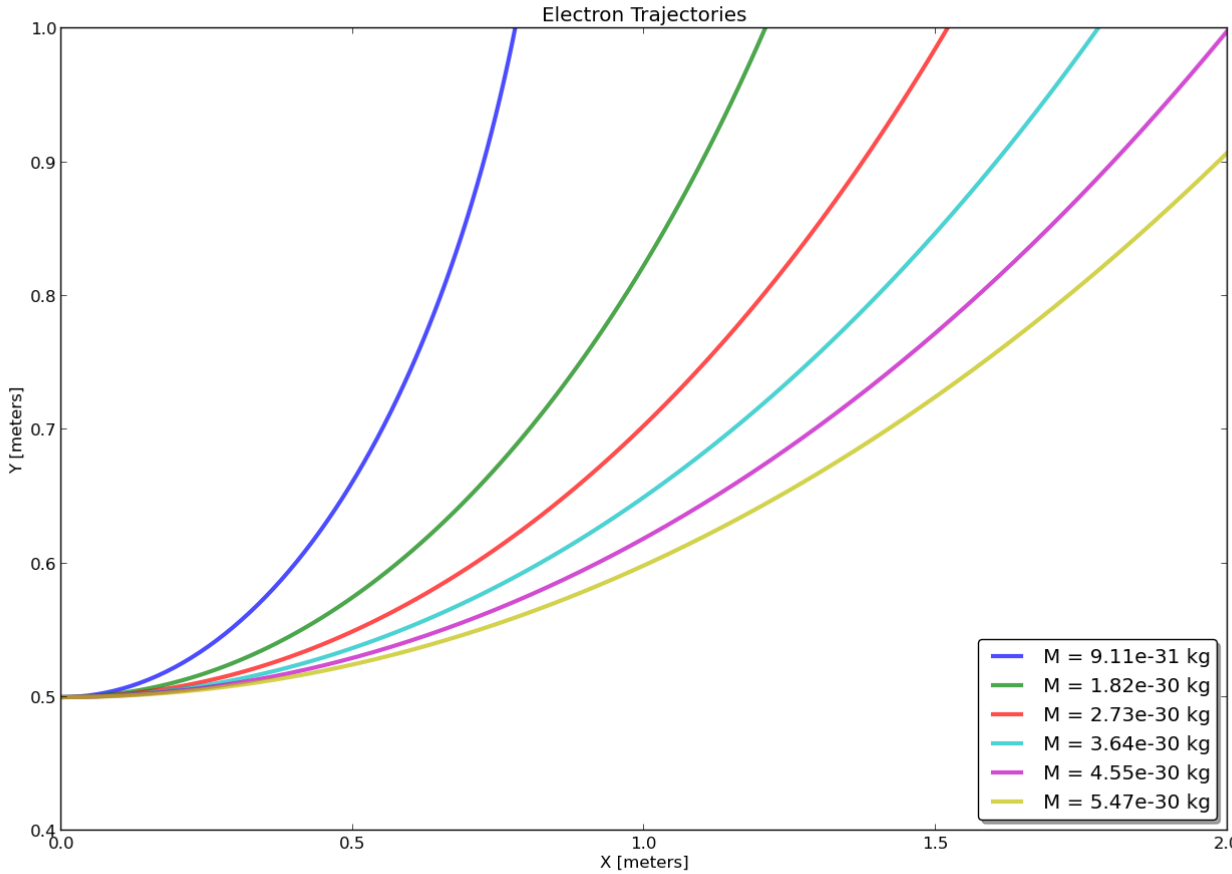
3.1 Build/Run

```
$ python electronTrajectory.py
```

3.2 Output

```
Starting calculation.
Calculating trajectory: 9.11e-31 kg
Calculating trajectory: 1.82e-30 kg
Calculating trajectory: 2.73e-30 kg
Calculating trajectory: 3.64e-30 kg
Calculating trajectory: 4.55e-30 kg
Calculating trajectory: 5.47e-30 kg
Plotting.
Saving plot to trajectory.png.
```

3.3 Plots



3.4 Methods

`electronTrajectory.`**`calculate_trajectory`**(*position*, *velocity*, *mass*, *B*)

Calculates the trajectory of the particle

Parameters

- **position** – 3D vector (r_x,r_y,r_z) in meters
- **velocity** – 3D vector (v_x,v_y,v_z) in m/s
- **mass** – scalar float in kg
- **B** – magnetic field strength, scalar float in Tesla

Returns a numpy array of 3D vectors (np.arrays)

`electronTrajectory.`**`main`**()

Loops over particles with integer multiples of the mass of the electron and shoots them through the magnetic field

`electronTrajectory.`**`plot_trajectory`**(*trajectories*, *masses*)

Creates a matplotlib plot and plots a list of trajectories labeled by a list of masses.

See Also:

called by `main()`

Parameters **trajectories** – an array of trajectories

:param masses:: a list of masses :returns: `None`

`electronTrajectory.`**`update_pos`**(*position*, *velocity*, *mass*, *B*)
calculates the magnetic force on the particle and moves it accordingly

Parameters

- **position** – 3D numpy array (r_x,r_y,r_z) in meters
- **velocity** – 3D numpy array (v_x,v_y,v_z) in m/s
- **mass** – scalar float in kg
- **B** – magnetic field strength, scalar float in Tesla

Returns the updated position and velocity (3D vectors)

PYTHON MODULE INDEX

d

doublePendulum, 6

e

electronTrajectory, 12

o

orbital, 3

p

plotting, 8

INDEX

A

animate() (in module plotting), 8
animate_paths() (in module plotting), 8

C

C1() (in module doublePendulum), 6
C2() (in module doublePendulum), 7
calculate_forces() (in module orbital), 3
calculate_paths() (in module doublePendulum), 7
calculate_trajectory() (in module electronTrajectory), 12
calculate_trajectory() (in module orbital), 3

D

deriv() (in module doublePendulum), 7
doublePendulum (module), 6
dp1() (in module doublePendulum), 7
dp2() (in module doublePendulum), 7
dtheta1() (in module doublePendulum), 7
dtheta2() (in module doublePendulum), 8

E

electronTrajectory (module), 12
euler() (in module doublePendulum), 8

F

force_gravity() (in module orbital), 3

M

main() (in module electronTrajectory), 12

O

orbital (module), 3

P

plot_orbit() (in module orbital), 4
plot_paths() (in module plotting), 9
plot_radii() (in module orbital), 4
plot_trajectory() (in module electronTrajectory), 12
plotting (module), 8

U

update_pos() (in module electronTrajectory), 13
update_pos() (in module orbital), 4

V

velocity_verlet() (in module doublePendulum), 8