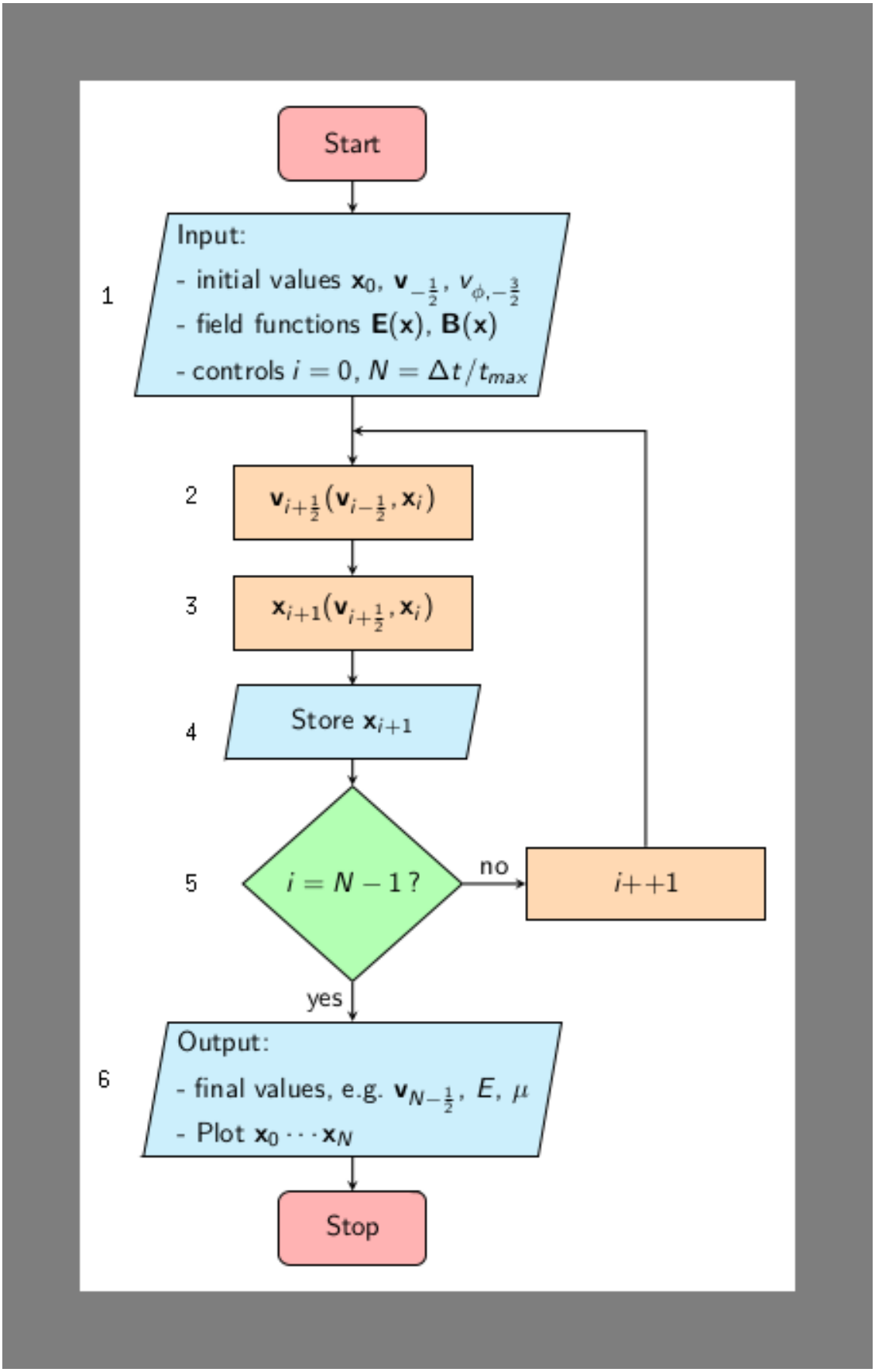
Untitled1

March 8, 2020

1 Computing particle orbits assignment

Bart van Pelt (0900221)

The goal is to implement the Boris scheme for particles moving in EM fields. I chose to work in a Jupyter notebook with IPython since these combine Python with easy plotting and text editing. The Boris scheme is shown in the figure below, where I have added some labels which I reference in my code. My implementation differs slightly from the scheme as shown in the assignment in that I do not explicitly initialize $v_{\phi,-\frac{3}{2}}$ instead, its value is taken from $v_{\phi,-\frac{1}{2}}$, since this is all that is asked. Also, I did not take N as an input, but instead $t_{\max}$, if one wants to calculate for N iterations it is trivial to use `EM_integrator(..., dt, N*dt)`. The integrator only outputs the computed trajectory of the particle, since the other outputs are trivial to derive from the trajectory and knowledge of the fields.


Start
Input:
- initial values $\mathbf{x}_0$, $\mathbf{v}_{-\frac{1}{2}}$, $v_{\phi,-\frac{3}{2}}$
- field functions $\mathbf{E}(\mathbf{x})$, $\mathbf{B}(\mathbf{x})$
- controls $i = 0$, $N = \Delta t / t_{max}$
1
2
$\mathbf{v}_{i+\frac{1}{2}}(\mathbf{v}_{i-\frac{1}{2}}, \mathbf{x}_i)$
3
$\mathbf{x}_{i+1}(\mathbf{v}_{i+\frac{1}{2}}, \mathbf{x}_i)$
4
Store $\mathbf{x}_{i+1}$
5
$i = N - 1$?
no
i++1
yes
Output:
- final values, e.g. $\mathbf{v}_{N-\frac{1}{2}}$, E, μ
- Plot $\mathbf{x}_0 \cdots \mathbf{x}_N$
6
Stop

Although the assignment suggests not to use arrays for vectors, the ease of using Numpy's vector operations convinced me otherwise. I am sticking to the $(\vec{e}_R, \vec{e}_Z, \vec{e}_\phi)$ basis, but for readability I will unpack the vectors where individual components are used, `P` is short for Phi in this context. In the loop I compute `v_new`/$\vec{v}_{+\frac{1}{2}}$ from `v_last`/$\vec{v}_{-\frac{1}{2}}$ and `v_P_old`/$v_{\phi,-\frac{3}{2}}$, the updating of these variables happens only in the very last lines of the loop. At some points in my code you will find a `# debug` comment, the lines following this are usually me deepcopying arrays and checking if they have changed. I do this because I ran into a really nasty bug because I did not take into account that numpy arrays are passed by reference, and the values I was passing into functions were actually changed in those functions. This was countered by deepcopying data wherever it may be needed, in some places I may have exaggerated a bit, but right now my code appears to be working as it should. You may also notice me using `cp()` to copy arrays as I pass them to functions, this was all done for debugging purpose, just know that I expect all my functions and variables to behave as if they were passed by value, no parameters should be changed!

1.1 Exercise 1

The annotated code is shown in the cell below, the comments with `step (N)` reference the annotations made in the figure above.

[1]:
```
# imports:
# some natural constants, useful for alpha mass and elemental charge
import scipy.constants as const
# numpy handles my vectors and the trajectory array
import numpy as np
# plotting library
import matplotlib.pyplot as plt
# configure it to display plots nicely in this notebook
%matplotlib inline
# very basic math
from math import ceil, pi

from copy import deepcopy as cp

# a function to integrate particle motion in an EM field
def EM_integrator(x_start, # initial position vector
                  v_start, # velocity vector minus half
                  E_field, # vector -> vector function
                  B_field,
                  dt,
                  t_max):
    # debug
    x0 = cp(x_start)
    v0 = cp(v_start)

    # some parameters I do not expect to change
    m = const.physical_constants['alpha particle mass'][0]
    q = 2*const.elementary_charge
    tau = q*dt/(2*m)
```

```
    # step (1), initialization
    # set the time to zero
    t = 0
    # the initial position is x0
    pos = x0
    # the initial velocity
    v_last = v0
    # the phi component of the velocity at t = -3/2
    v_P_old = v_last[2]
    # compute how many steps are needed
    N = ceil(t_max/dt)
    # initialize an array to store the position data
    trajectory = np.empty([3, N])

    # a loop, Pythonic notation for (5) and the i++ blocks
    for i in range(N):
        # new timestep
        t += dt

        # working up to step (2) by calculating v_min, B_star and v_plus

        # compute v_minus as in the assignment
        v_min = v_last + tau*E_field(cp(pos))

        # unpack the speed and position vectors for readability
        v_R, v_Z, v_P = v_last
        x_R, x_Z, x_P = pos

        # B_star is just the magnetic field at the current position...
        B_star = B_field(cp(pos))
        #...but with this term subtracted from the Z component...
        B_star[1] -= (1.5*v_P - 0.5*v_P_old)*m/(q*x_R)
        #...and multiplied by this factor
        B_star *= q*dt/m

        # c1,2,3 as defined in the assignment, using Numpy's dot products here␣
↪for ease
        c_1 = 4/(4+B_star.dot(B_star))
        c_2 = 2*c_1 - 1
        c_3 = 0.5*c_1*v_min.dot(B_star)

        # adding the different contributions together as v_plus
        v_plus  = c_1*np.cross(v_min, B_star)
        v_plus += c_2*v_min
        v_plus += c_3*B_star
```

```
        # step (2), v_new is finally calculated
        v_new = v_plus + tau*E_field(cp(pos))

        # step (3), calculate the new position
        pos += v_new*dt

        # step (4), and store it
        trajectory[:, i] = pos

        # update the variables for the next iteration
        v_P_old = v_P
        v_last  = v_new

    # debug
    # commented out the x0 one because that is actually okay, x_start is free␣
↪to change, x0 should not change
    # if not np.array_equal(x0, x_start): print('x0 changed in EM_int')
    if not np.array_equal(v0, v_start): print('v0 changed in EM_int')

    # step (6), return the trajectory
    return trajectory
```

1.2 Exercise 2

[2]:
```
# get the mass and charge of an alpha particle, only SI units are used
mass   = const.physical_constants['alpha particle mass'][0] # kg
charge = 2*const.physical_constants['elementary charge'][0] # C

# set the kinetic energy to 3.5MeV
energy = 3.5e6*const.physical_constants['electron volt'][0] # J
# compute the speed from kinetic energy and mass
speed  = (2*energy/mass)**0.5 # m/s

# set tokamak parameters
R_0 = 1 # m
B_0 = 5 # T
B_p0 = 1 # T

# compute the gyro-frequency and gyro-radius
gyro_frequency = B_0*charge/(2*pi*mass)  # Hz
gyro_radius    = mass*speed/(charge*B_0) # m

# and display them
print(f'The alpha particle has a gyro-frequency of {gyro_frequency:.3} Hz')
print(f'and a gyro-radius of {gyro_radius:.3} m')
```

```
The alpha particle has a gyro-frequency of 3.84e+07 Hz
and a gyro-radius of 0.0539 m
```

1.3 Exercise 3

Here I'm defining the electric and magnetic fields. Note that B depends on parameters R_0, B_0 and B_{p0}, while the `EM_integrator` needs a vector -> vector function. To encapsulate this I define a parametrized `B_param`, which I do not directly pass into `EM_integrator`, but instead I make a modified function, in which the parameters are taken care of, which does have the right function signature.

[3]:
```
# in this exercise the electric field function should return a zero vector
def E_0(pos):
    return np.array([0, 0, 0])

# the parametrized magnetic field
def B_param(pos, R_m, B_m, B_pm):
    # debug
    pos_cp = cp(pos)
    # unpack the position vector for readability
    x_R, x_Z, x_P = pos

    # calculate r using numpy's hypot function
    r = np.hypot(x_R-R_m, x_Z)

    # calculate the B components
    B_R = B_pm*x_Z*R_m/(r*x_R)
    B_Z = B_pm*(R_m-x_R)*R_m/(r*x_R)
    B_P = B_m*R_m/x_R

    # debug
    if not np.array_equal(pos_cp, pos): print('B_param modifies pos')

    # assemble them in to a proper array and return
    return np.array([B_R, B_Z, B_P])

# handle the parameters, and make a function with the proper signature
B_field = lambda pos: B_param(pos, R_0, B_0, B_p0)
```

With the fields defined we can start looking at particle trajectories. Start with setting the initial conditions.

[4]:
```
# position
x_1 = np.array([1.85*R_0, 0, 0])

# velocity direction
v_dir = np.array([0, 0.6, -0.8])
```

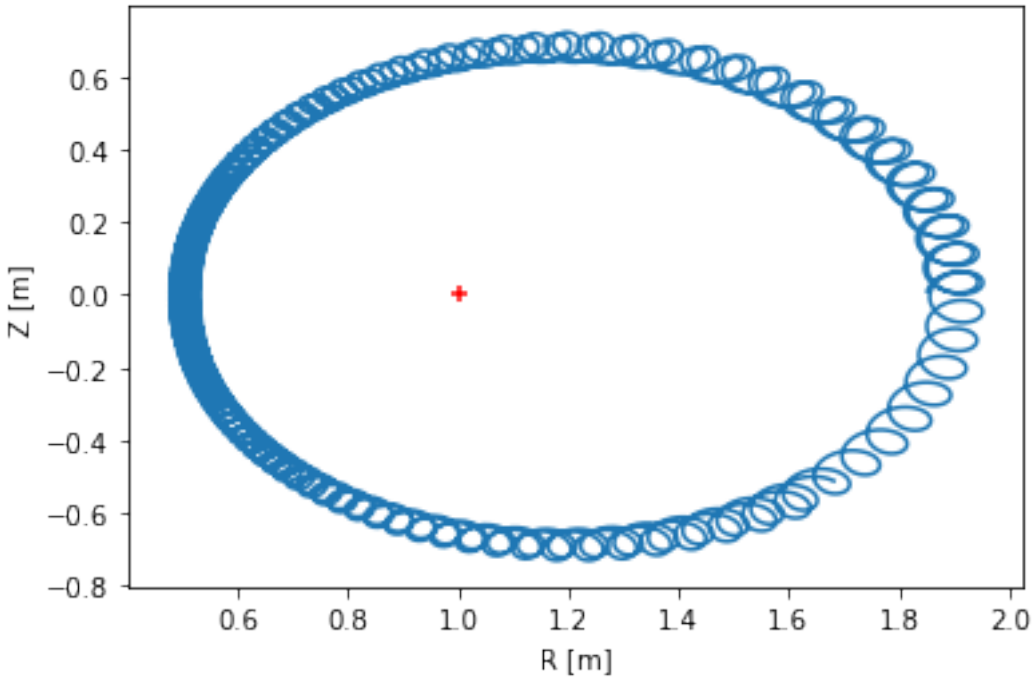
```
# just a quick check to ensure the direction is indeed a unit vector
# this is just a warning though, normalization should be done by the user
if v_dir.dot(v_dir) != 1: print('Warning: v_dir is not normalized')

# regardless of a possible warning, continue by scaling v_dir with the wanted␣
 ↪magnitude
v_1 = speed*v_dir
```

Getting all the plotting commands in one function, could be useful later on. I chose some default values, but these can obviously be changed. The `t_max` value was chosen because it appears the particle is able to perform a few full poloidal turns for this value.

[5]:
```
def RZ_plot(x_0=x_1,
            v_0=v_1,
            E_field=E_0,
            B_field=B_field,
            dt = 0.05/gyro_frequency,
            t_max  = 5e-6,
            R_0 = R_0):
    # compute the trajectory for 2000 time steps given IC's, fields
    traj = EM_integrator(x_0, v_0, E_field, B_field, dt, t_max)

    # unpack trajectory array for readability
    R = traj[0,:]
    Z = traj[1,:]
    # and do some plotting
    fig, ax = plt.subplots()
    # plot the trajectory
    ax.plot(R, Z)
    # put a red marker at the magnetic center
    ax.scatter([R_0],[0], c='red', marker='+')
    # add labels
    ax.set_xlabel('R [m]')
    ax.set_ylabel('Z [m]')
    # and return the trajectory in case the user wants to perform further␣
 ↪analysis
    return traj
```

Use time step $0.05\Omega_c^{-1}$

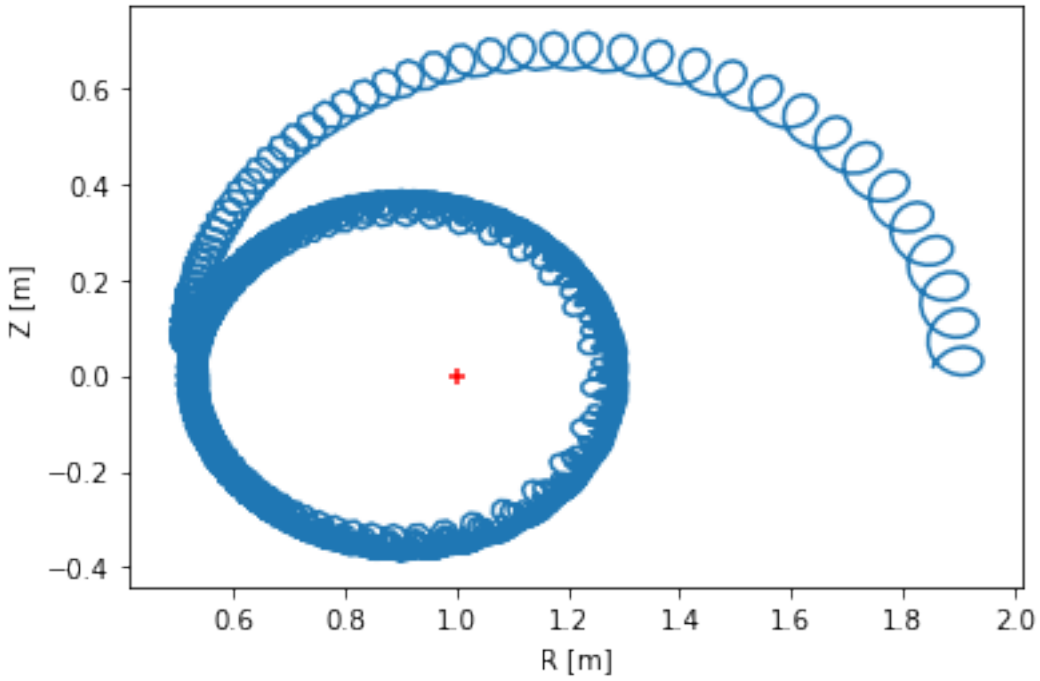
[6]:
```
traj1 = RZ_plot(dt=0.05/gyro_frequency)
```



Now let's experiment a bit with the dt value, I'm just showing a couple examples.

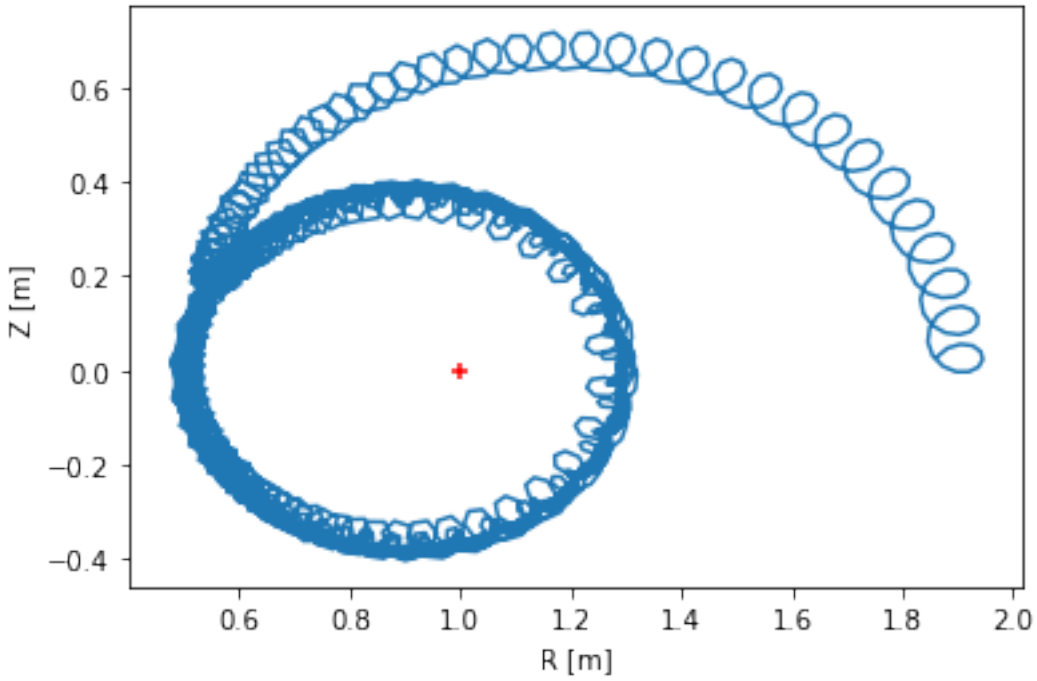
[7]:
```
RZ_plot(dt=0.10/gyro_frequency, t_max = 5e-5) # somewhat okay, starts to drift␣
↪gradually for large t
```

[7]:
```
array([[ 1.85529899e+00,  1.86527277e+00,  1.87875613e+00, …,
         1.05524617e+00,  1.04861684e+00,  1.04193405e+00],
       [ 1.94125479e-02,  3.61967724e-02,  4.89514189e-02, …,
        -3.30622252e-01, -3.33440313e-01, -3.36192438e-01],
       [-2.72243826e-02, -5.48813318e-02, -8.31940713e-02, …,
         5.81794134e+02,  5.81827213e+02,  5.81860286e+02]])
```



[8]:
```
RZ_plot(dt=0.15/gyro_frequency, t_max = 5e-5) # this gets weird
```

[8]:
```
array([[ 1.86150982e+00,  1.88160338e+00,  1.90527261e+00, …,
         1.06000169e+00,  1.04986315e+00,  1.03968655e+00],
       [ 2.75367763e-02,  4.69381460e-02,  5.51487583e-02, …,
        -3.44508099e-01, -3.48715981e-01, -3.52618960e-01],
       [-4.10847901e-02, -8.34940497e-02, -1.27664895e-01, …,
         5.95172471e+02,  5.95222052e+02,  5.95271649e+02]])
```

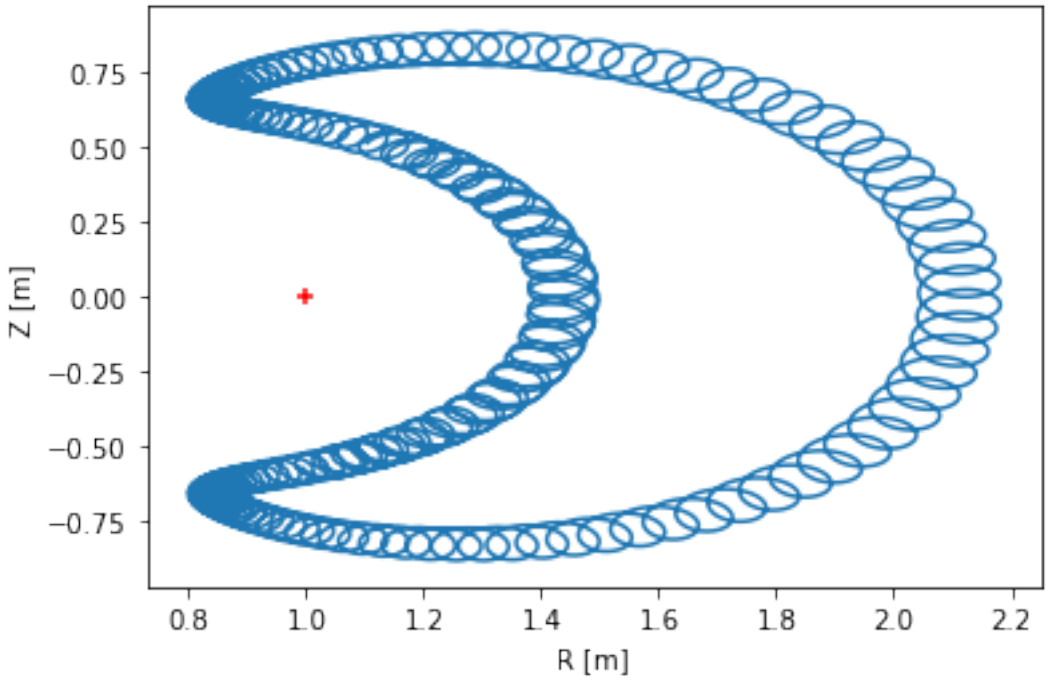


Stuff appears to diverge for dt>0.1/gyro_frequency, so I'll stay at 0.05/gyro_frequency.

1.4 Exercise 4

I can just reuse the `RZ_plot()` function defined above.

[9]:
```
# make a plot of the trajectory
x_2 = np.array([1.1*R_0, 0.5*R_0, 0])
v_2 = np.array([0, 0.8, 0.6])*speed
traj2 = RZ_plot(x_0 = x_2,
        v_0 = v_2,
        t_max = 5e-6)
```



Now let's make a function that determines the magnetic moment from the velocity and magnetic field vectors, the function also needs the mass, but since we're usually only considering alpha particles the mass is set to a default alpha particle mass. Also a function that evaluates B_max is defined, although I'm not sure how this should be done exactly, since a derivation was only worked out for simple magnetic mirrors, and I'm not sure how B_{max} and B_{min} can be calculated in the case of a tokamak. Freidberg only appears to refer to these parameters in the context of the slab model for the magnetic field (Figure 8.12). Nonetheless, I'm trying to do something close to slide 21 of the lecture on orbits, but the definition of r is unclear to me.

[10]:
```
def magnetic_moment(velocity_in, B_vector_in, mass=mass):
    # debug
    velocity = cp(velocity_in)
    B_vector = cp(B_vector_in)

    # compute the magnitude of B
    B_magnitude = np.linalg.norm(B_vector)
    # normalize B to get direction vector
    B_direction = B_vector/B_magnitude
    # compute v component parallel to B
    v_parallel  = velocity.dot(B_direction)
    # v^2 = v_perp^2 + v_parr^2 -> v_perp^2 = v^2 - v_parr^2
    v_perp_sq   = velocity.dot(velocity) - v_parallel**2

    # debug
```

```
    if not np.array_equal(velocity_in, velocity): print('velocity changed in␣
 ↪magnetic_moment')
    if not np.array_equal(B_vector_in, B_vector): print('velocity changed in␣
 ↪magnetic_moment')

    # equation for magnetic moment
    return mass*v_perp_sq/(2*B_magnitude)

# function that should calculate B_max, this is probably not right though
def B_max(x_0_in, B_field = B_field, R_0 = R_0):
    # debug
    x_0 = cp(x_0_in)

    # unpack position vector
    x_R, x_Z, x_P = x_0
    # calculate distance from magnetic center
    r = np.hypot(x_R - R_0, x_Z)
    # position of point where B_max should be evaluated
    pos_B_max = np.array([R_0 - r, 0, 0])

    # debug
    if not np.array_equal(x_0, x_0_in): print('x_0 changed in B_max')

    # norm of B_field(..) at this point
    return np.linalg.norm(B_field(pos_B_max))
```

Use these to check the passing criterion $\epsilon > \mu B_{max}$

[11]:
```
print(f'Particle 1 passing: {energy > magnetic_moment(v_1,␣
 ↪B_field(x_1))*B_max(x_1)}')
print(f'Particle 2 passing: {energy > magnetic_moment(v_2,␣
 ↪B_field(x_2))*B_max(x_2)}')
```

```
Particle 1 passing: False
Particle 2 passing: False
```

This agrees with the numerical results.

1.5 Exercise 5

Let's make a function that computes the electric field.

[12]:
```
def E_rad_param(pos_in, R_0, E_0):
    # debug
    pos = cp(pos_in)

    # unpack
    R, Z, P = pos
```

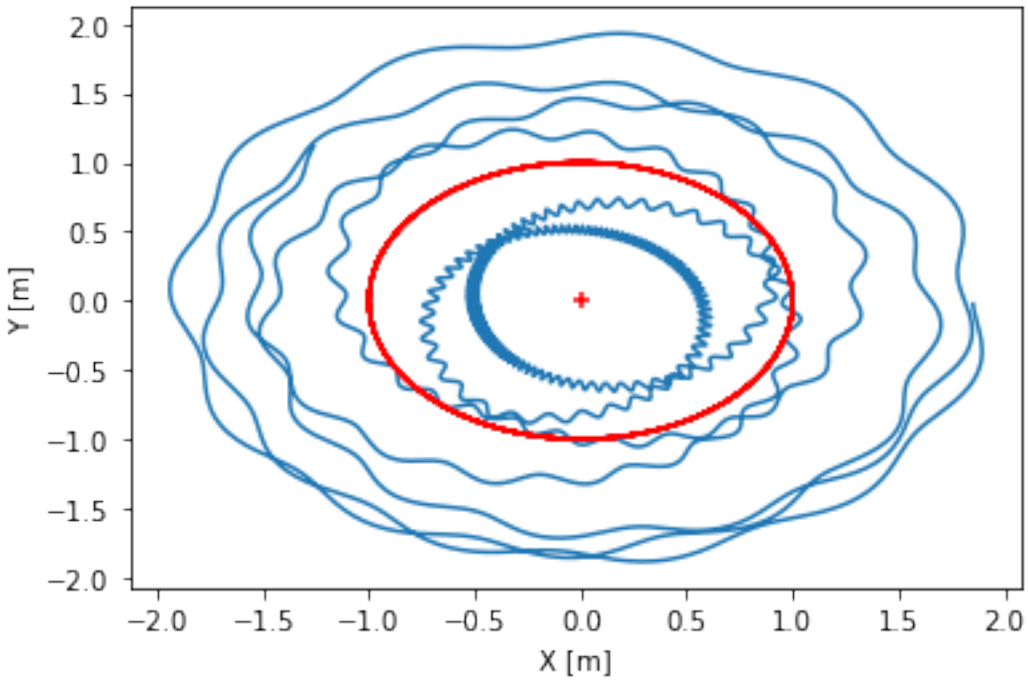
```
    # compute components
    E_R = E_0*(R-R_0)/R
    E_Z = E_0*Z/R
    E_P = 0

    # debug
    if not np.array_equal(pos, pos_in): print('E_rad_param changes pos')

    # assemble vector and return
    return np.array([E_R, E_Z, E_P])
```

A function to make a top plot would be nice:

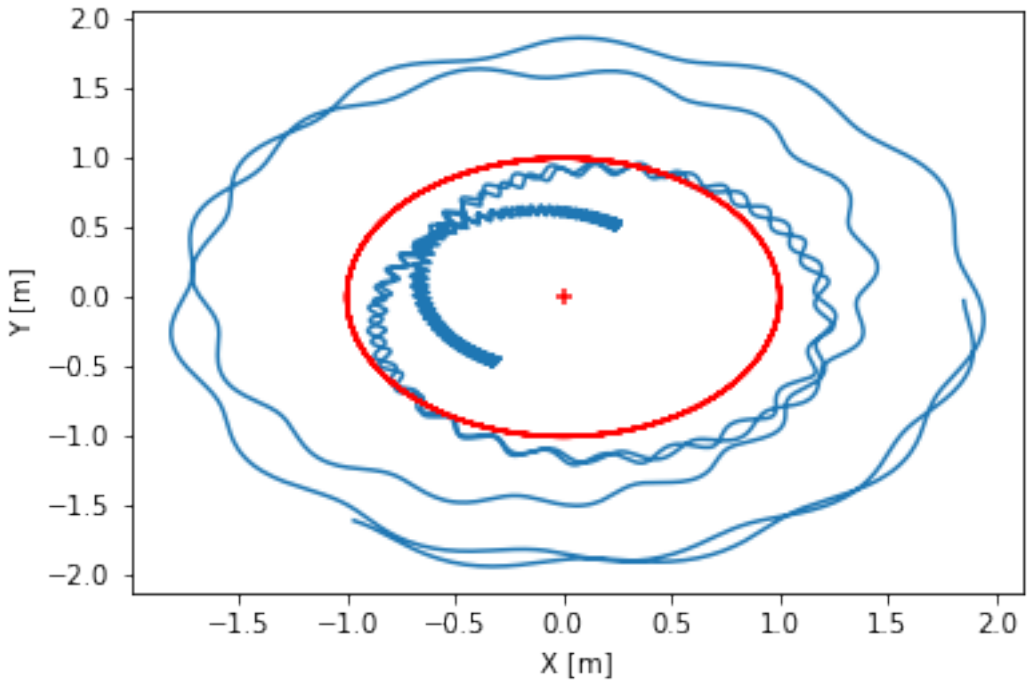
[34]:
```
def top_plot(traj_in, R_center=R_0):
    # debug
    traj = cp(traj_in)

    # unpack coordinates
    R = traj[0,:]
    Z = traj[1,:]
    P = traj[2,:]
    # compute X and Y
    C = np.cos(P)
    S = np.sin(P)
    X = np.multiply(R, C)
    Y = np.multiply(R, S)
    # and do this for plotting the magnetic center line
    X_center = R_center*C
    Y_center = R_center*S


    # and do some plotting
    fig, ax = plt.subplots()
    # plot the trajectory
    ax.plot(X, Y)
    # put a red marker at the center of the tokamak
    ax.scatter([0],[0], c='red', marker='+')
    # and plot the magnetic center
    ax.plot(X_center, Y_center, c='red')
    # add labels
    ax.set_xlabel('X [m]')
    ax.set_ylabel('Y [m]')

    # debug
    if not np.array_equal(traj, traj_in): print('top_plot changes traj')
```

Playing with E_0 for particle 1.

[36]:
```
# first with no electric field, for reference
E_rad = lambda pos: E_rad_param(pos, R_0, 0)
traj = EM_integrator(x_1, v_1, E_rad, B_field, dt = 0.05/gyro_frequency, t_max
 ↪= 5e-6)
top_plot(traj, R_center = R_0)
```

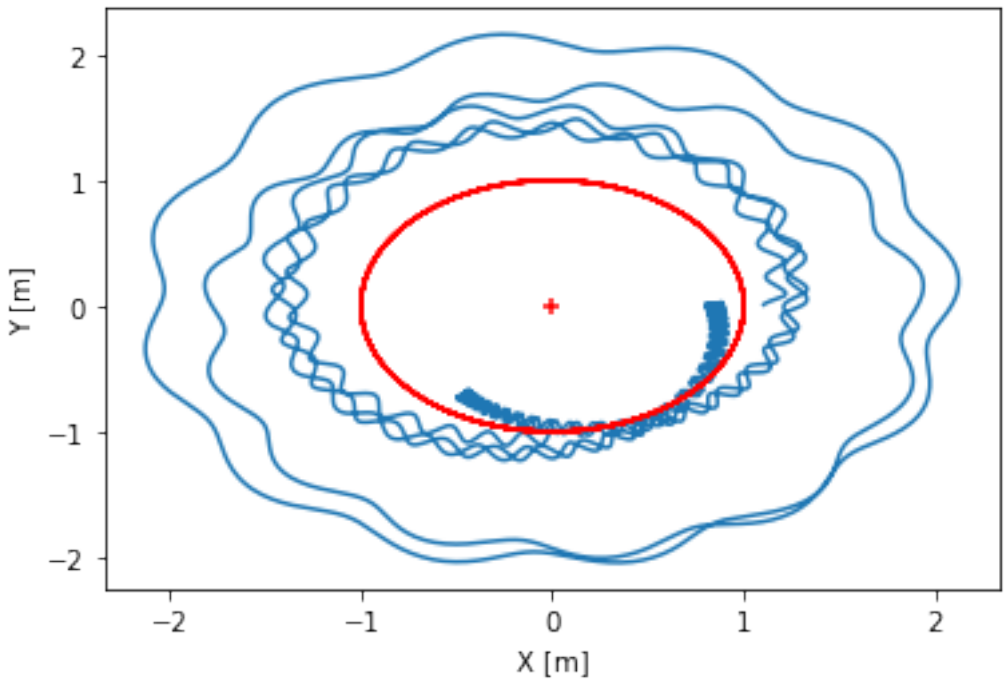


[44]:
```
# with a very high electric field
E_rad = lambda pos: E_rad_param(pos, R_0, 1000000)
traj = EM_integrator(x_1, v_1, E_rad, B_field, dt = 0.05/gyro_frequency, t_max
 ↪= 5e-6)
top_plot(traj, R_center = R_0)
```

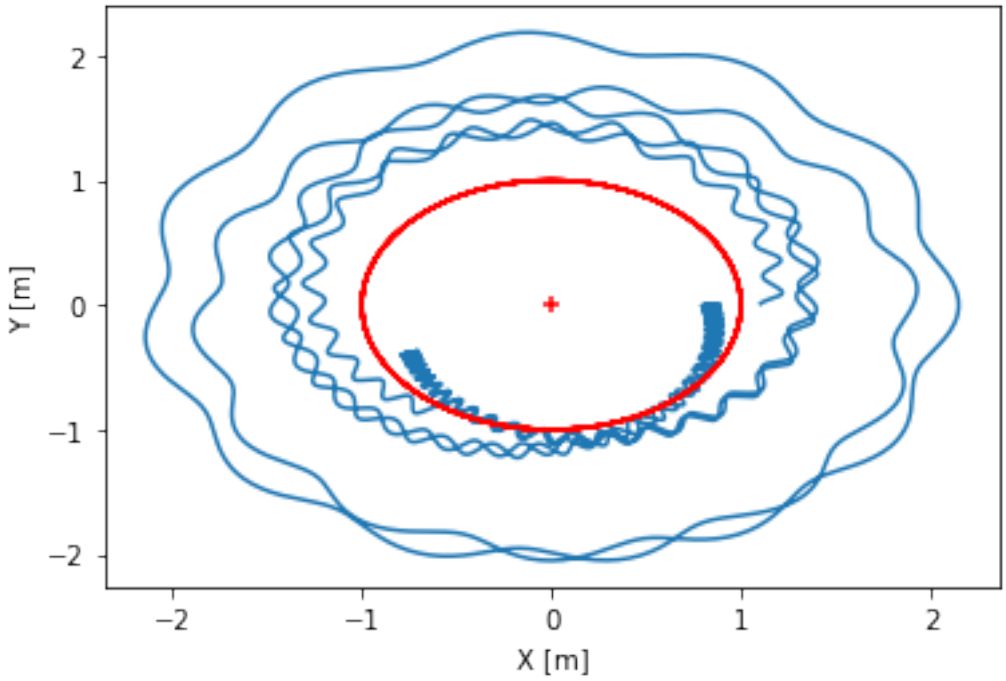


And for particle 2:

[38]:
```
# first without electric field, for reference
E_rad = lambda pos: E_rad_param(pos, R_0, 0)
traj = EM_integrator(x_2, v_2, E_rad, B_field, dt = 0.05/gyro_frequency, t_max␣
↪= 5e-6)
top_plot(traj, R_center = R_0)
```



[39]:
```
# with a very high electric field
E_rad = lambda pos: E_rad_param(pos, R_0, 100000)
traj = EM_integrator(x_2, v_2, E_rad, B_field, dt = 0.05/gyro_frequency, t_max␣
↪= 5e-6)
top_plot(traj, R_center = R_0)
```

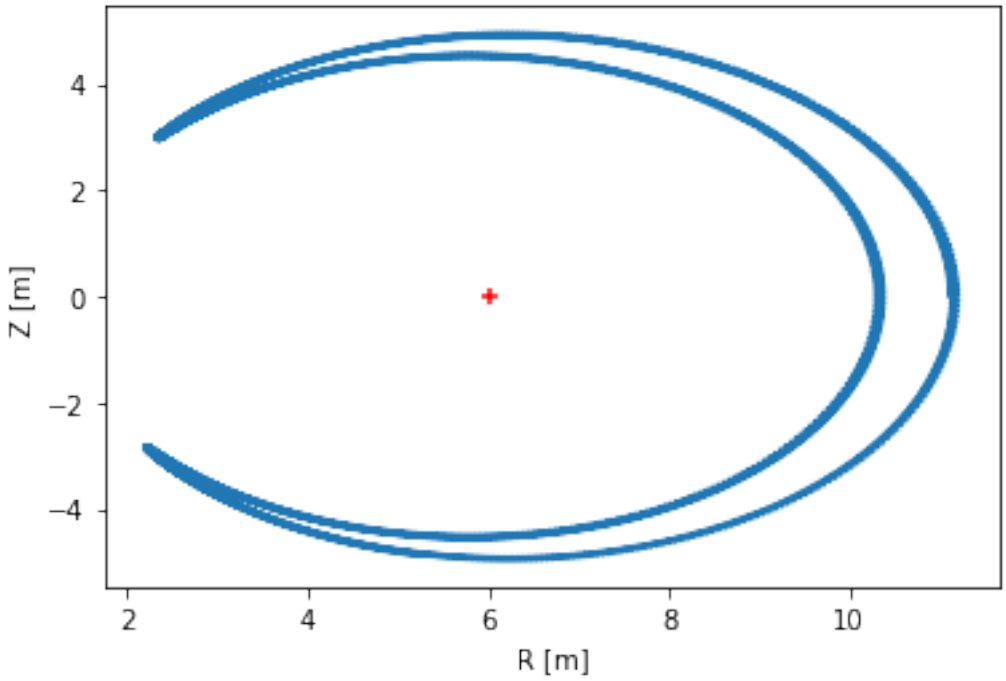


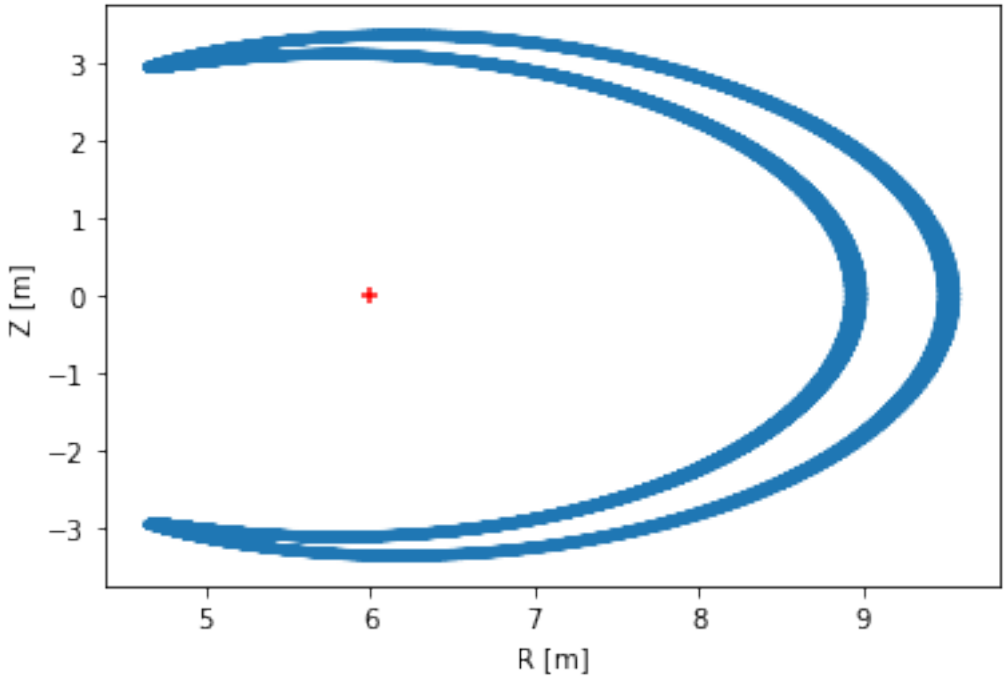
From the top plots I did not notice any really interesting changes for differing E_0

1.6 Exercise 6

In this case I just need to change the R_0 that goes into the fields, and change the initial conditions.

[18]:
```
R_ITER = 6 # m
B_ITER = lambda pos: B_param(pos, R_ITER, B_0, B_p0)
# simply scale the initial coordinates with R_ITER/R_0
x_1_ITER = x_1*R_ITER/R_0
x_2_ITER = x_2*R_ITER/R_0

# particle 1
_ = RZ_plot(x_1_ITER, v_1, B_field=B_ITER, t_max = 5e-5, R_0 = R_ITER)

# particle 2
_ = RZ_plot(x_2_ITER, v_2, B_field=B_ITER, t_max = 5e-5, R_0 = R_ITER)
```


4
2
0
−2
−4
Z [m]
2
4
6
8
10
R [m]


3
2
1
0
−1
−2
−3
Z [m]
5
6
7
8
9
R [m]