Magnetic Mirror Effect in Magnetron Plasma: Modeling of Plasma Parameters

Different aspects of the program are discussed based on the different notebooks (ipynb files) that describe them.

GitHub repository:

https://github.com/18BME2104/MagneticMirror

Required functionalities and Files:

1. Particle - **particle.ipynb**
2. Electric and Magnetic fields - **field.ipynb**
3. Particle initialization - **sampling.ipynb**
4. Updating the particles - **step.ipynb**
5. Batches of updates - **run.ipynb**
6. Plotting - **plot.ipynb**

1 Constants - constants.ipynb

The **constants.ipynb** notebook describes some constants useful in the program. Some useful constants are e (electron charge) m_e (electron mass), charges and masses of ions in the plasma, $a.m.u$ (atomic mass unit), N_A (Avogadro's number), ϵ_0 (permittivity of vacuum), μ_0 (permeability of vacuum), K or k_B (Boltzmann's constant), etc.

```
1  class Constants:
2      def __init__(self):
3          self.constants = {
4              #'symbol': ['value', unit', 'digits', '10 ^ power', 'number of
                   digits', 'description'],
5               }
6      def show_constant(self, symbol):
```

2 Particle - particle.ipynb

The **particle.ipynb** notebook describes the state of a particle; its position, velocity, mass, charge, name, and optionally acceleration (which is set to 0 as default if it is not required

to track the acceleration of a particle) as of now.

Currently the **Boris algorithm** is an update strategy defined to update the state of a particle. Other strategies could be defined in new functions in the class. However, Boris algorithm is good enough for us to get started.

```
class Particle:
    #... other things

    def Boris_update(self, afield, argsE, argsB):
        # Define q_prime
        q_prime = (self.charge / self.mass) * (dt / 2)

        # Get E and B fields from the afield argument by passing in the
            current position of the particle
        argsE = V, center
        E = afield.get_E_field(self.r, V, center)
        argsB = n, I, R, B_hat, mu_0
        B = afield.get_B_field(self.r, n, I, R, B_hat, mu_0)

        #Boris velocity update
        v_minus = self.v + q_prime * E
        v_plus = v_minus + q_prime * 2 * np.cross(v_minus, B)
        v_new = v_plus + q_prime * E

        self.v = v_new


        #could have also done:
        #self.v += (2 * q_prime) * (E + np.cross( (self.v + q_prime * E),
            B))

        #update position
        self.r += v_new * dt

    #... some other things
```

3 Electric and Magnetic fields - field.ipynb

Electric and Magnetic field configurations described in **field.ipynb**.

Currently **Uniform Electric field** and the **Radial Electric field** (the field depends on the particle's position) created by an electrode are available to set up electric fields. **Uniform Magnetic field** and Magnetic field created by a **Helmholtz coil** and that by two Helmholtz coils are available.

Other Electric and Magnetic fields can be defined as functions in this class. These fields could be based on modeling of apparatus used to create electric or magnetic fields such as coils, or based on analytic expressions.

```python
class Field:
    #... something

    def uniform_E_field(self, E):
        return E

    def radial_E_field(self, r, V, center = [0,0,0]):
        #Get the distance vector of the particle from the electrode
        dr = [(r[0] - center[0]), (r[1] - center[1]), 0]
        #Get the electric field
        E = V * dr
        return E

    def uniform_B_field(self, B):
        return B

    def helmholtz_coil_B_field(self, n, I, R, B_hat, mu_0):
        return ( (4/5)**1.5 * ( (mu_0 * n * I) / (R) ) * B_hat)

    def two_helmholtz_B_field(self, n1, I1, R1, B1_hat, n2, I2, R2, B2_hat
        , mu_0):
        B1 = helmholtz(self, n1, I1, R1, mu_0, B1_hat)
        B2 = helmholtz(self, n2, I2, R2, mu_0, B2_hat)

        #Calculate the resultant of two magnetic fields
        B_hat = B1_hat + B2_hat
        return B_hat

    #...something else
```

4 Particle initialization - sampling.ipynb

The initial positions and velocities of the particles play an important role in the evolution of their state under the influence of electric and magnetic fields. The initial distribution of positions and velocities define where the particles start in the setup (or lab apparatus); for example where they are injected into a sputtering chamber through valves, and what velocities they start with; for example what potential they are accelerated through or what parameters were used for the pumps used to pump the particles in.

The tracking of initial distribution is also important in determining how the magnetic mirror effect is observed in the plasma. The Sampler class samples initial positions and velocities for a given number of particles based on some available schemes.

Currently positions can be sampled such that all particles start at the **same position** (like for example if injected through a port), **at the same distance** but randomly distributed in angular positions relative to some point(origin as of now). Velocities can be sampled such that particles have the **same speed**, **same velocity** or **Maxwellian distributed speeds** or **Maxwellian distributed velocities**.

```python
class Sampler:
    #... something

    def sample_same_given_position(self, r, n):
        positions = []
        for i in range(n):
            positions.append(r)

        return np.array(positions)

    def sample_same_given_distance_all_random_direction(self, d, n):
        positions = []

        for i in range(n):
            positions.append(d * uniform_random_unit_vector())

        return np.array(positions)

    def sample_same_given_velocity_same_direction(self, v, n):
        velocities = []
        for i in range(n):
            velocities.append(v)

        return np.array(velocities)

    def sample_same_given_speed_all_random_direction(self, s, n):
        velocities = []

        for i in range(n):
            velocities.append(s * uniform_random_unit_vector())

        return np.array(velocities)

    def sample_velocity_uniformKE_same_given_direction(self):
        pass

    def sample_velocity_uniformKE_all_random_directions(self, n):
        pass

    def sample_Maxwellian_speed(self, v_median, K, T, m, n):
        alpha = math.sqrt(K * T / m)
        speeds = stats.maxwell.rvs(loc = v_median, scale = alpha, size = n
            )
        return speeds

    def sample_Maxwellian_velocity_same_given_direction(self, v_median, K,
        T, m, v_hat, n):
        speeds = sample_Maxwellian_speed(self, v_median, K, T, m, n)
        velocities = np.outer(speeds, v_hat)

        return np.array(velocities)

    def sample_Maxwellian_velocity_same_random_direction(self, v_median, K
```

```
            , T, m, n):
52          speeds = sample_Maxwellian_speed(self, v_median, K, T, m, n)
53          direction = self.uniform_random_unit_vector()
54          velocities = np.outer(speeds, direction)
55
56          return np.array(velocities)
57
58      def sample_Maxwellian_velocity_all_random_direction(self, v_median, K,
            T, m, n):
59          speeds = sample_Maxwellian_speed(self, v_median, K, T, m, n)
60          velocities = []
61          for i in range(n):
62              velocities.append(speeds[i] * np.array(
                    uniform_random_unit_vector()))
63
64          return np.array(velocities)
65
66      def sample_parabolic_velocity(self):
67          pass
68      #... something else
```

The initial positions and velocities could be passed around in lists but to be able to reuse some generated samples, and avoid sampling all the time, it is convenient to write sampled positions and velocities to files (csv format seems convenient).

```
1   class Sampler:
2       #... something
3       def write_to_csv_file(self, ...):
4
5           # write array to csv file
6
7       def write_file_name(self, ...):
8
9           # write the file name to the list of available files
10
11      #... something else
```

5 Updating the particles - step.ipynb

The Step class is concerned with **initializing particles**, **defining the fields**, and **updating the states of the particles** (positions and velocities) under the action of electric and magnetic fields.

```
1   class Step:
2
3       # ... something
4
5    ### PARTICELS INTIALIZATION SECTION
6       def initialize_particles(self, names, q_s, m_s, r_0_s, v_0_s, a_0_s, n
            ):
7
8           for i in range(n):
```

```
            self.particles.append(Particle(names[i], q_s[i], m_s[i], r_0_s
                [i], v_0_s[i], a_0_s[i] ))

### FIELDS INITIALIZATION SECTION
    def initialize_fields(self):
        self.fields = Field()

### TIME STEP SECTION
    def update_particles(self, dt, argsE, argsB):

        for particle in self.particles:
            particle.update(self.fields, dt, argsE, argsB)
    # ... something else
```

To use sampled positions and velocities that have been saved in csv files, some reading functionality is useful to load these positions and velocities to be used during initialization.

```
class Step:

    # ... something
    def read_r_or_v_file_and_reshape(self, index):

        array_from_file = self.read_r_or_v_file(index)
        reshaped_array = self.reshaper(array_from_file)
        return reshaped_array

    # ... something else
```

6 Batches of updates - run.ipynb

The Run class is concerned with running the steps defined by the Step class in step.ipynb. This includes creating batches of particles, updating them under the influence of electric and magnetic fields and removing them if they move out of the region of interest or for example are absorbed during a coating process. New batches of particles can be created to model the flow of particles as in a plasma chamber setup. Functionality to change the fields; for example changing the Voltages in electrodes or Currents in the coils, are also defined. This better models the control of plasma supply and electric and magnetic field control apparatus as in a laboratory.

```
class Run:

    # ... something
    def create_particles(self):
        pass

    def update_particles(self):
        pass

    def remove_particles(self):
```

```
11          # This might include particles moving outside the chamber (the
                Electric and Magnetic fields or
12          # simply positions of interest) or particles being absorbed for
                example in a coating process
13          pass
14
15      def create_fields(self):
16          pass
17
18      def change_fields(self):
19          pass
20
21      #... something else
```

7 Plotting - plot.ipynb

The plot class is used to define functionalities to draw particle positions, plot positions and velocities of particles, as the system evolves like a lab apparatus. These functionalities are imported into run.ipynb; and if required other notebooks, to be used when the simulation is running.

References

[1] Qin, H., Zhang, S., Xiao, J., & Tang, W. M. (April, 2013). *Why is Boris algorithm so good?*. Princeton Plasma Physics Laboratory, PPPL-4872.