run

April 9, 2022

[1]:
```
from inspect import getmembers, isfunction, isclass # To inspect imported␣
 ↪modules
from IPython.display import display, Math, Latex, clear_output # To use latex␣
 ↪typesetting style

import numpy as np # For computations
import sympy as sm # For symbolic use

import pandas as pd # To use large tables
import matplotlib.pyplot as plt #To make plots
import csv # To read csv file of position and velocities required to initialize␣
 ↪particles
import import_ipynb
from batch import Batch
```

```
importing Jupyter notebook from batch.ipynb
importing Jupyter notebook from particle.ipynb
importing Jupyter notebook from field.ipynb
```

[1]:
```
class Run:
    '''
    The class Run is an extension of the class Batch from the notebook batch.
 ↪ipynb
    The class Run creates different batches of particles that form the plasma,␣
 ↪and works with them
    '''

    def __init__(self):
        self.batches = []
        '''
        self.batches will hold dictionaries where each dictionary is like␣
 ↪{description: Batch object}
        '''
        self.descriptions = []
        '''
        self.descriptions holds the descriptions so that Batch objects␣
 ↪corresponding to descriptions
```

```
        can be retrieved from self.batches
        '''

    def __string__(self):
        pass

    def docstring(self):
        pass
        # print (something)

    def create_batch_with_file_initialization(self, name, charge, mass, N, n,␣
↪description, r_index = 0, v_index = 1):
        '''
        Initializes a batch using reading position and velocity data
        from saved files.

        It is assumed that all particles are of the same species and hence
        have the same charge and mass.

        Arguments:
        self
        name: Names of a particle in the batch to be created
        charge: Charge of a particle in the batch to be created
        mass: Mass of a particle in the batch to be created
        N: Number of files saved in the file
        n: Number of particles to be created in the batch
        description: Description of the batch, so that the batch can be␣
↪retrived later

        r_index = 0 #Load the first position file in the list of available␣
↪position files for default
        v_index = 1 #Load the second velocity file in the list of available␣
↪velocity files for default
        '''


        abatch = Batch()
        abatch_dict = dict()
        self.descriptions.append(description)
        abatch_dict[description] = abatch
        self.batches.append(abatch_dict)
        # first file with v_median = 800 m/s had issues so second file with␣
↪v_median = 0 might be better
        r0_s = abatch.read_r_file_and_reshape(r_index, N, n)
        r0_s = np.array(r0_s).astype(float)
        v0_s = abatch.read_v_file_and_reshape(v_index, N, n)
        v0_s = np.array(v0_s).astype(float)
```

```
        abatch.initialize_particles_of_same_species(name, charge, mass, r0_s,␣
↪v0_s, n)

    def create_batch_with_saved_file(self, names, charges, masses, n):
        pass

    def display_available_batches(self):
        pass

    def update_batch_with_unchanging_fields(self, index, dT, stepT, argsE,␣
↪argsB, track):
        '''
        Updates a batch of particles, under the condition
        that the electric and magnetic fields are time independent (unchaging).

        Arguments:
        self
        index: The index of the batch to be updated.
        OR also The index of description in self.descriptions,
        corresponding to the batch to be updated,
        which is the value corresponding to the key description from self.
↪descriptions[index]
        dT: Duration of update
        stepT: Duration of single step of the update
        argsE: arguments required for the electric field
        argsB: arguments required for the magnetic field
        track: list of particles in this batch that is to be updated to be␣
↪tracked
        For example: [1,30,18] means
        track the 1st 30th and 18th particles in this batch

        Returns:
        positions_and_velocities: A dictionary whose keys are elements of track␣
↪i.e. the particles to be tracked.
        and keys are lists of tuples (time step i, particle position after ith␣
↪time step, particle velocity after ith update)
        '''

        positions_and_velocities = dict()
        for a_index in track:
            positions_and_velocities[a_index] = []
        nT = int(dT / stepT)
        description = self.descriptions[index]
        batch_to_update = self.batches[index][description] #This is a batch␣
↪object
```

```
        for i in range(nT):
            position_and_velocity = batch_to_update.update_particles(stepT,␣
↪argsE, argsB, track)

            for a_index in track:
                positions_and_velocities[a_index].append((i,␣
↪position_and_velocity[a_index][0], position_and_velocity[a_index][1]))

        return positions_and_velocities

    def modify_batch_description(self):
        pass

    def save_batch_to_file(self):
        # Use description for the batch to pass as details for␣
↪write_to_csv_file method for Batch class object.
        #while writing
        pass

    def remove_batch(self):
        # This might include particles moving outside the chamber (the Electric␣
↪and Magnetic fields or
        # simply positions of interest) or particles being absorbed for example␣
↪in a coating process
        pass

    def remove_particles_from_batch(self):
        pass

    def create_fields(self):
        pass

    def change_fields(self):
        pass
```

[ ]: