Simulation of Charged Particles Motion in Electromagnetic Field

Stefano Markidis

KTH Royal Institute of Technology

Equation of Motion for Charged Particles

$$\frac{d\mathbf{p}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) + \mathbf{F}_{non-EM}$$

- p = momentum = m γ v (we will assume non-relativistic motion γ = 1)
- q = particle charge
- E = electric field acting on the particle
- B = magnetic field on the particle
- $F_{non\text{-}EM}$ = other forces acting on particle (gravitational, …)

Solving the Equation of Motion with Python

- Python NumPy for arrays.
- Python `SciPy.Integrate` provides:
  - Efficient ODE solvers are already implemented.
- Python `matplotlib`
  - Plotting subroutines.


```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
```

ODE to Solve

$$\frac{d\mathbf{p}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) + \mathbf{F}_{non-EM}$$

Input:

- `t`: Current time (required by solve_ivp, but unused here).
- `x_vect`: State vector `[x,y,z,u,v,w]`.
- Magnetic Field Calculation:
  - `Bx, By, Bz`: Magnetic field components in the Earth dipole field. There is no electric field in this example.
- Derivatives:
  - Position derivatives: `[dxdt, dydt, dzdt]` are the velocity components.
  - Velocity derivatives: `[dudt, dvdt, dwdt]` come from the Lorentz force:
- Divided by mass `m` to get acceleration.

- **Output:** the derivatives `[dxdt, dydt, dzdt, dudt, dvdt, dwdt]` for use by the solver (solve_ivp).

```
# Newton-Lorentz equation function
def newton_lorentz(t, x_vect):
  # Unpack position and velocity
  x, y, z, u, v, w = x_vect
  # Magnetic field components
  fac1 = -B0 * Re**3 / (x**2 + y**2 + z**2)**2.5
  Bx = 3 * x * z * fac1
  By = 3 * y * z * fac1
  Bz = (2 * z**2 - x**2 - y**2) * fac1
  # Charge-to-mass ratio
  qom = q / m
  # Derivatives
  dxdt = u
  dydt = v
  dzdt = w
  dudt = qom * (v * Bz - w * By)
  dvdt = qom * (w * Bx - u * Bz)
  dwdt = qom * (u * By - v * Bx)
  return [dxdt, dydt, dzdt, dudt, dvdt, dwdt]
```

SciPy ODE Solver

```
# Solve the ODE
solution = solve_ivp(newton_lorentz, [0, tfin], initial_conditions, t_eval=time, method='RK45')
```

- `solve_ivp`: A function from `scipy.integrate` to solve initial value problems for ordinary differential equations (ODEs).
  - Numerically integrates the system of ODEs defined in `newton_lorentz` over the specified time span.
- **Parameters:**
  - `newton_lorentz`: The function that defines the system of ODEs.
  - `[0, tfin]`: The time span for the solutions
  - `initial_conditions`: The initial values of the state vector:
    - [x0, y0, z0, u0, v0, w0] (position and velocity components).
  - `t_eval=time`: Time points where the solution is evaluated.
  - time: Array of time steps.
  - method='RK45': Solver method:
    - Runge-Kutta (4th and 5th order) adaptive step-size solver.
- **Output:**
  - `solution.t`: The time points (from t_eval).
  - `solution.y`: The solution values (state variables at each time point).

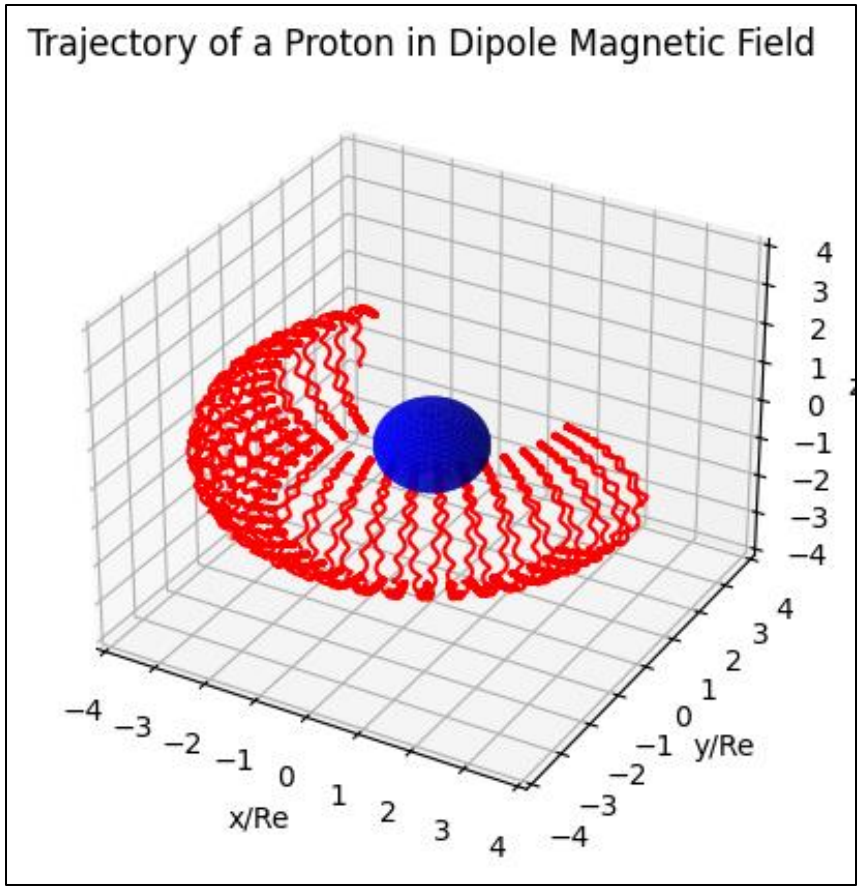
Main Code

```
# Parameters for the proton trajectory
K = 1e7 * e # Kinetic energy in Joules
v_mod = c / np.sqrt(1 + (m * c**2) / K) # Speed
# Initial position: equatorial plane 4Re from Earth
x0 = 4 * Re
y0 = 0
z0 = 0
# Initial velocity
pitch_angle = 30.0 # degrees
u0 = 0.0
v0 = v_mod * np.sin(np.radians(pitch_angle))
w0 = v_mod * np.cos(np.radians(pitch_angle))
# Initial conditions
initial_conditions = [x0, y0, z0, u0, v0, w0]
# Time span
tfin = 80 # Final time in seconds
time = np.arange(0, tfin, 0.01) # Time array

# Solve the ODE
solution = solve_ivp(newton_lorentz, [0, tfin], initial_conditions,
t_eval=time, method='RK45')

# Extract the solution
x_sol = solution.y.T # Transpose for easier handling (time steps in rows)
# Plotting the trajectory
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(x_sol[:, 0] / Re, x_sol[:, 1] / Re, x_sol[:, 2] / Re, 'r')
ax.set_xlabel('x/Re')
ax.set_ylabel('y/Re')
ax.set_zlabel('z/Re')
ax.set_title('Trajectory of a Proton in Dipole Magnetic Field')
plt.show()
```

Define the Parameters

ODE Solver

Plotting



Result of the Simulation