EM Bonus HW Note

Paul Tan

April 2018

I would like to extend my thanks to Mr Gabriel Wu, who suggested the RK4 method in solving the problem numericallly, Mr Zaw Lin Htoo, who helped me out on the atan function and provided me great insight over email and Dr Koh, who suggested including markers to improve presentation.

You can access the animation at `https://sorewachigauyo.github.io/penning-trap/main.html`

This note will be going over the core code mainly in main.js and Particle.js

1 Particle Object

We first describe the particle's parameters (position, velocity, etc.) through the use of an object.

Listing 1: Object constructor Particle.js (lines 4-11)

```
function Particle(data){
    this.mass = 0;
    this.charge = 0;
    this.position = [0,0,0];
    this.accel = [0,0,0];
    this.velocity = [0,0,0];
    this.initVoltage = 0;
    this.dsqrd = 0;
```

Since this is a simulation, we will have to update the particle's parameters, this is done at every time interval. We first update the acceleration based on the Lorentz force, followed by the velocity and finally the position.

Listing 2: Updating parameters Particle.js (lines 26-33)

```
Particle.prototype.updateParameters = function(timeInterval = 1, bField = [0,0,0]){
    const eField = quadrupoleEField(this.initVoltage,this.position,this.dsqrd);
    this.accel = vecScalDiv(this.calcLorentzForce(eField,bField,this.velocity),
        this.mass);
    this.updateVelocity(timeInterval,bField);
    this.updatePosition(timeInterval,bField);
}
```

We get electric force component by the quadrupole electric field. We are given that the quadrupole potential in cylindrical coordinates is determined by

$$V(\mathbf{r}) = \frac{V_0}{2d^2}\left(z^2 - \frac{\rho^2}{2}\right) \tag{1}$$

Then,

$$E = -\nabla V = -\frac{V_0}{2d^2}(2z\hat{z} - \rho\hat{\rho}) \tag{2}$$

Listing 3: Electric field function Quadrupole.js

```
function quadrupoleEField(initVoltage, position, dsqrd){
    const cylin=getCylindricalCoords(position);
    const r=cylin[0];
    const theta = cylin[1];
    const z=cylin[2];
    let Ecylin = vecScalMultiply([-r,0,2*z],-initVoltage/(2*dsqrd));
    Ecylin[1] = theta;
    return fromCylinToCartesian(Ecylin);
}
```

The Lorentz force gives us the following equation which we use to determine the particle's acceleration, Particle.js (lines 37-41).

$$F = q(E + v \times B) \tag{3}$$

Next, to determine the velocity and the position, we use the 4th order Runge-Kutta method (RK4).

Suppose we have an initial value problem,

$$y' = f(t, y), \quad y(t_0) = y_0$$

and

$$t_{n+1} = t_n + h$$

Then, we have

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where h is the time interval,

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

This lets us construct the next value y_{n+1} through a sort of trapezoid. The values provided by f are obtained through Euler's method. We use RK4 in functions updateVelocity and updatePosition and Euler's method in getNextAccel or getNextVelocity for the values of f.

2 Animation script

Now that we have an idea of how the particle changes through each time division, we are ready to begin animating the particle. Most of the code in main.js is setting up the animation overlay and particle parameters but we are most interested in the animation loop.

We create a Particle object called particle to contain our kinematic variables and our sphere to be animated, called sphere. It is then assumed that the viewer is running the animation at 60 frames per second.

Listing 4: Start of animation loop and marker main.js (lines 59-71)

```
function animate() {
    requestAnimationFrame( animate );

    counter++;
    if (counter % 10 === 0){

        let marker = new THREE.Mesh(markGeom.clone(), markMat);
        marker.position.copy(sphere.position);
        scene.add(marker);
        marker.name = counter;
    }
```

Before this code, we have initialized a counter to keep track of the number of frames animated. requestAnimationFrame creates a new frame. Every ten frames, we create a marker at the particle's current position. Then, we add it to the scene to be animated. After 2400 frames or 240 markers, markers, starting from the first, will be removed every 10 frames (the same rate as markers are added).

The next block of code in main.js updates the html page with the particle's kinematic parameters every second.

Listing 5: Updating particle parameters main.js (lines 96-98)

```
    for (let i=0; i<10; i++){
        particle.updateParameters(1/600,bField);
    }
```

For every frame, we compute the particle's parameters 10 times (with the time division adjusted appropriately, or 1/600 second^{-}1). This is done to increase the accuracy of our simulation as we are solving the system numerically.

Finally, we update sphere's position based on our particle object. and render it as a frame.

This animation loop continues until the user closes the tab.

3 References

The kinematic equations affecting the particle was obtained and derived from the introduction given in the preamble in `http://gabrielse.physics.harvard.edu/gabrielse/papers/1990/1990_tjoelker/chapter_2.pdf`

Rounding function was obtained from `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round`

Trajectory function obtained from `https://stackoverflow.com/questions/18370701/how-to-draw-the-trajectory-of-a-mesh-with-threejs`

Runge-Kutta method inspired from `https://en.wikipedia.org/wiki/Runge\OT1\textendashKutta_methods`

Marker removal function from `https://stackoverflow.com/questions/39322054/three-js-remove-specific-object-from-scene`