

Electricity and Magnetism

Numerical Analysis of Wien Filter Velocity Selector

Amit Roth
Amit Harel

1. Introduction

We will solve at first the following problem.

Suppose we have a charged particle with a charge q , moving under the influence of a constant electric field and a constant magnetic field.

$$\begin{cases} \vec{E}(\vec{r}) = E\hat{y} \\ \vec{B}(\vec{r}) = -B\hat{x} \end{cases}$$

In $t = 0$, the particle has the following velocity, $v(t = 0) = u\hat{z}$ and $u = \frac{3E}{B}$.

Solution

Let's start calculating the force on such a particle. The magnetic field points in \hat{x} and the velocity in \hat{z} . We can reduce our problem in to 2-D problem on the zy plane. We will get:

$$\begin{cases} \vec{v} = v_y\hat{y} + v_z\hat{z} \\ \vec{B} = -B\hat{x} \end{cases} \Rightarrow q\vec{v} \times \vec{B} = qv_yB\hat{z} - qv_zB\hat{y}$$

So the net force will be

$$\vec{F} = (qE - qv_zB)\hat{y} + qv_yB\hat{z}$$

And from newton's second law

$$\begin{cases} m\dot{v}_y = qE - qv_zB \\ m\dot{v}_z = qBv_y \end{cases}$$

We can derive from the second equation

$$m\ddot{v}_z = qB\dot{v}_y \Rightarrow \dot{v}_y = \frac{m}{qB}\ddot{v}_z$$

We will substitute our result for \dot{v}_z in the first equation and we will get

$$\frac{m^2}{q^2B^2}\ddot{v}_z = \frac{E}{B} - v_z$$

We got an harmonic oscillator for v_z , the solution is

$$v_z = A\cos(\omega t + \phi) + \frac{E}{B}$$

We know that $v(t = 0) = u\hat{z}$, and we know that $\dot{v}_z(t = 0) = 0$. So the full solution for v_z is

$$v_z = \frac{2E}{B}\cos\left(\frac{qB}{m}t\right) + \frac{E}{B}$$

From here we can derive v_y instantly using:

$$v_y = \frac{m}{qB}\dot{v}_z = \frac{m}{qB}\frac{-2E}{B}\frac{qB}{m}\sin\left(\frac{qB}{m}t\right)$$

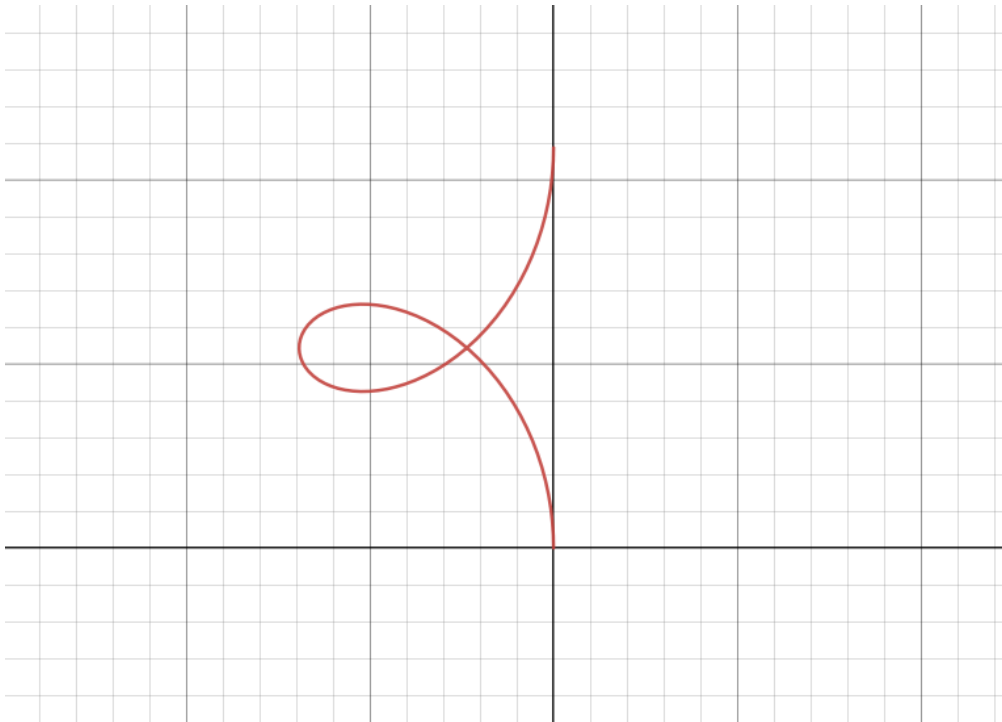
$$\Rightarrow v_y = \frac{-2E}{B}\sin\left(\frac{qB}{m}t\right)$$

Now, in order to get r_y and r_z we will just integrate the velocities, using the given $\vec{r}(t = 0) = 0$.

$$r_y = \int v_y dt = \frac{2mE}{qB^2} \cos\left(\frac{qB}{m}t\right) - \frac{2mE}{qB^2}$$

$$r_z = \int v_z dt = \frac{2mE}{qB^2} \sin\left(\frac{qB}{m}t\right) + \frac{E}{B}t$$

The graph for $\vec{r}(t)$ for $0 \leq t \leq \frac{2\pi}{\omega}$ is



2. Numerical Integration

Taylor First Order

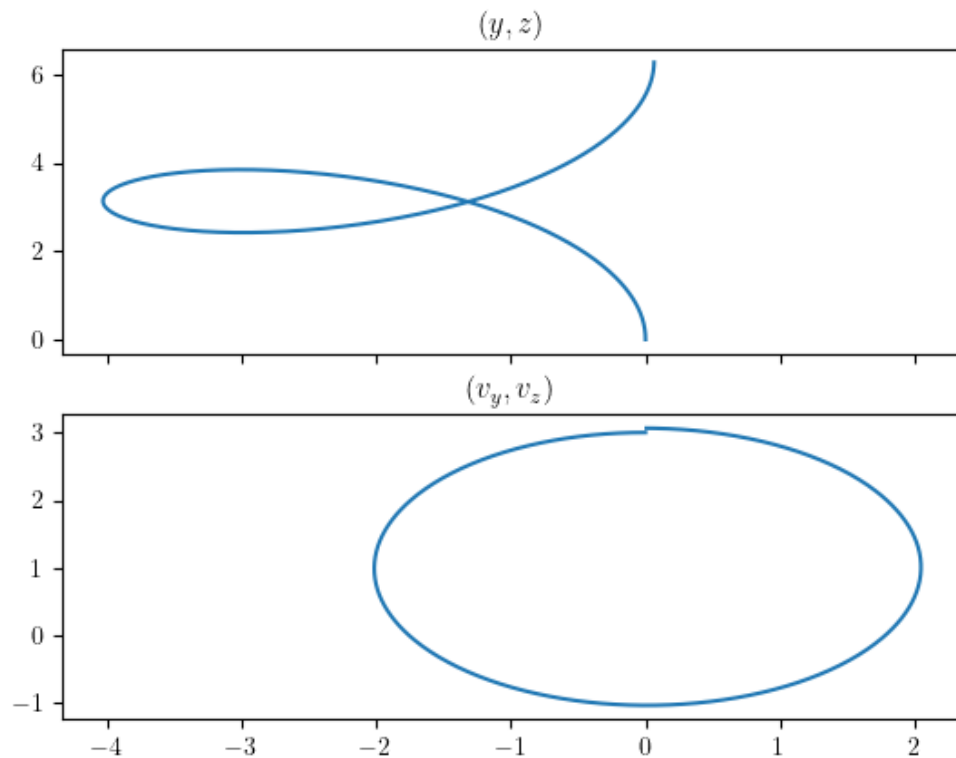
We will now solve our problem using numerical methods. Using first order Taylor approximation we will get the following relations:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \frac{d\mathbf{r}}{dt}\Delta t + O(\Delta t^2) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + O(\Delta t^2)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{d\mathbf{v}}{dt}\Delta t + O(\Delta t^2) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t + O(\Delta t^2)$$

The code is implemented in the file `numerical_integration.py`, and also appended below in the code appendix.

The output graphs are:



We can see that the graph (y, z) has the same form as the graph that we got from the analytical solution.

Midpoint

We will now use a more precise approximation. Using the midpoint technique, with:

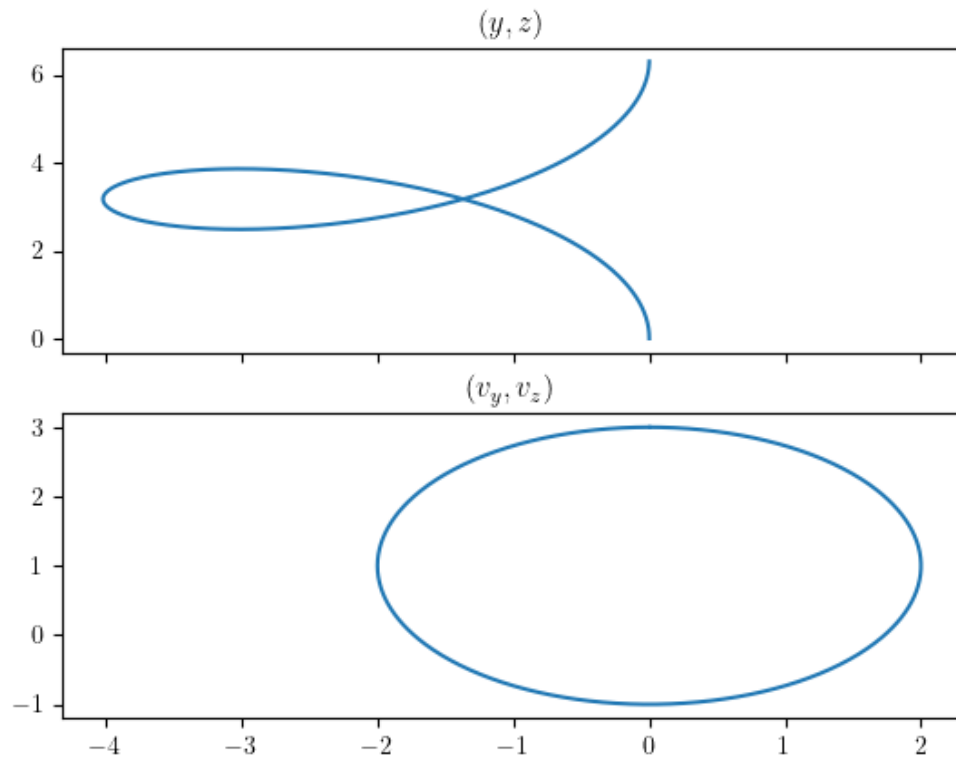
$$\begin{cases} f(t, y) = \frac{\partial f}{\partial t} \\ k_1 = \Delta t \cdot f(t, y(t)) \\ k_2 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_1}{2})) \end{cases}$$

And we will calculate the result using

$$y_{n+1} = y_n + k_2 + O(\Delta t^3)$$

The code is implemented in the file `numerical_integration.py`, and also appended below in the code appendix.

The output graphs are here:



We can see that the graphs are smoother.

Runge-Kutta

A more accurate technique can be implemented using:

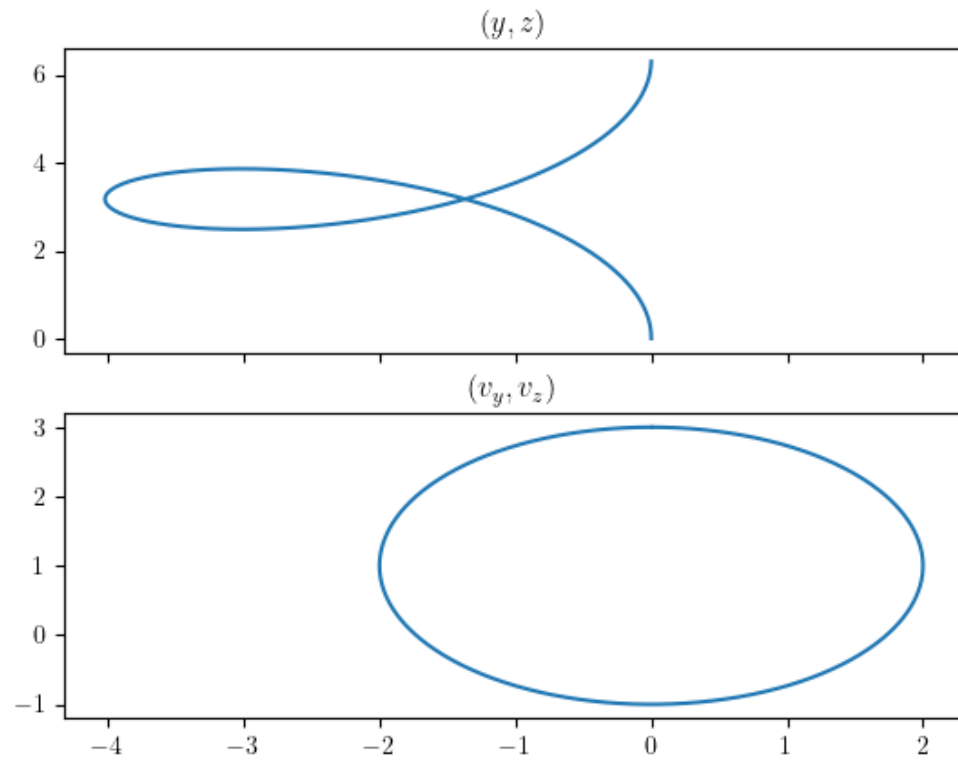
$$\begin{cases} k_1 = \Delta t \cdot f(t, y(t)) \\ k_2 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_1}{2})) \\ k_3 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_2}{2})) \\ k_4 = \Delta t \cdot f(t + \Delta t, y(t + k_3)) \end{cases}$$

And we will get

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\Delta t^5)$$

The code is implemented as well in the file `numerical_integration.py`, and also appended below in the code appendix.

The output graphs are similar to the midpoint graphs:



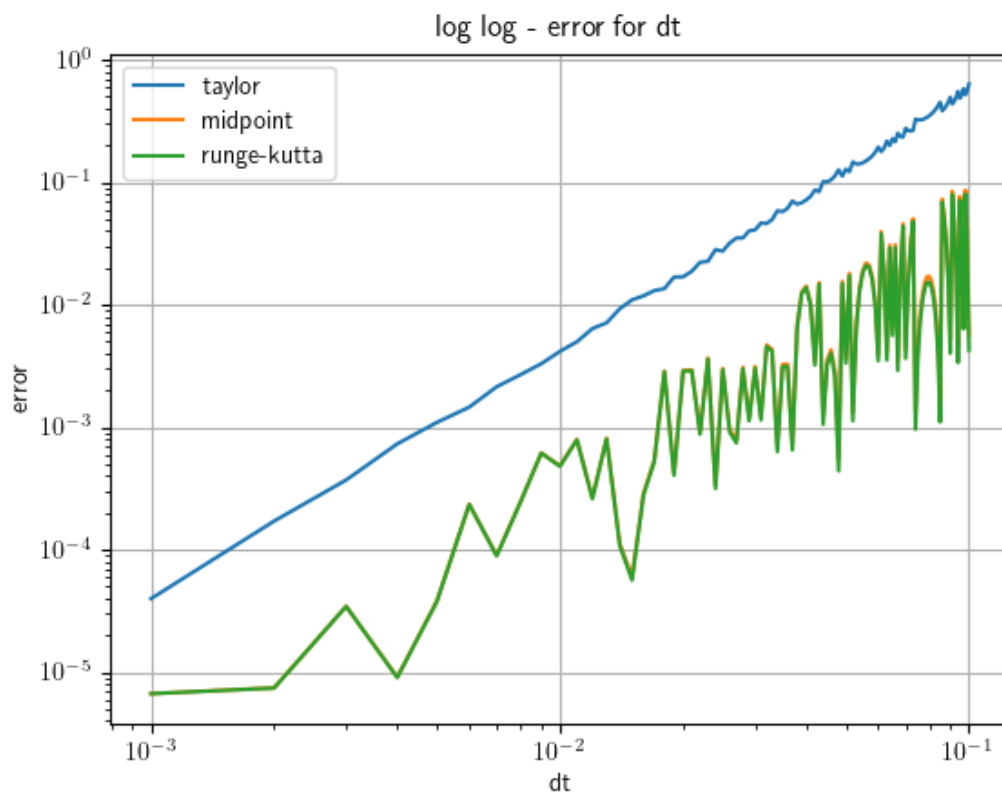
Benchmark

We will calculate the point that the particle will be in $T = \frac{2\pi}{\omega}$.

In the analytical solution we get:

$$\begin{cases} r_z(T) = \frac{2mE\pi}{qB^2} \\ r_y(T) = 0 \end{cases}$$

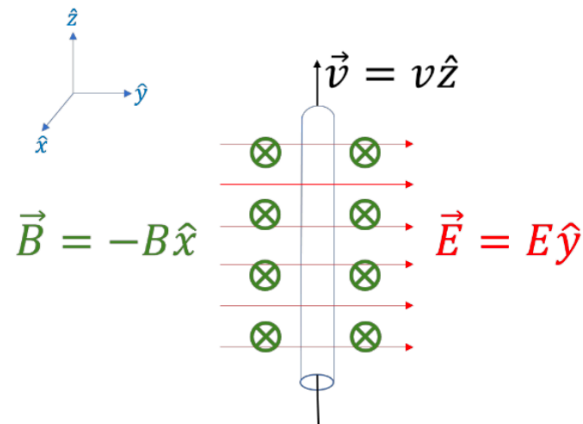
We will calculate the error (euclidian distance from the analytical to the numeric) and plot it against dt. The graph in log log scale is:



We can see that midpoint and range have similar results, but Taylor is much worse.

3. Wien Filter Velocity Selector

We will now deal with the following problem of Wien filter velocity selector.



We already solved the problem analytically in the first section.

Assume we have beam of protons traveling with average kinetic energy

$E_0 = 5MeV = 8.0109 \cdot 10^{-13}J$, and pipe of length $l = 1m$ and radius

$r = 3mm$.

The ratio $\frac{E}{B}$

The initial velocity we need to set in order to let the protons' beam to pass

can be derived instantly from our solution and is $v_0 = \frac{E}{B}\hat{z}$

Solving for the path of the particles

We will solve the same problem, but taking into consideration with the initial energy and the initial coordinates in the pipe.

Assume for all particles $E_{initial} \in [E_0 - \delta E, E_0 + \delta E]$ for $\delta E = 0.25[MeV]$ and $E_0 = 5[MeV]$. We can derive the velocity from the energy using

$$v_0 = \sqrt{\frac{2E_0}{m}}. \text{ We will get:}$$

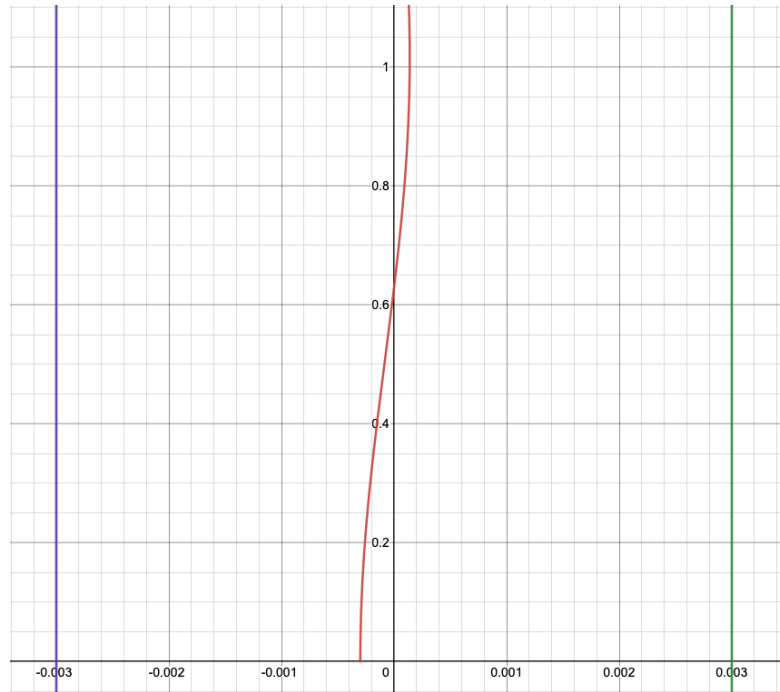
$$\begin{cases} v_z = (v_0 - \frac{E}{B})\cos(\frac{qB}{m}t) + \frac{E}{B} \\ v_y = (\frac{E}{B} - v_0)\sin(\frac{qB}{m}t) \end{cases}$$

And after integration,

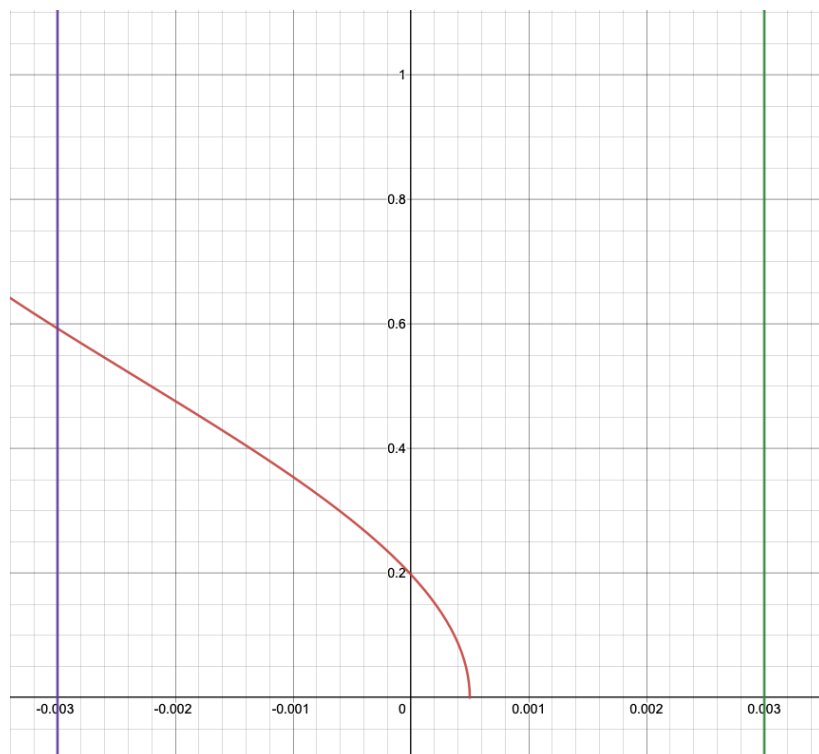
$$\begin{cases} r_z = (v_0 - \frac{E}{B})\frac{m}{qB}\sin(\frac{qB}{m}t) + \frac{E}{B}t \\ r_y = \frac{m}{qB}(v_0 - \frac{E}{B})(\cos(\frac{qB}{m}t) - 1) + y_0 \end{cases}$$

Before we will show the numerical solution, we plotted the analytical solution in desmos. The link to the graph is in the appendix, it is very beautiful to change the parameters and observe how the graph changes.

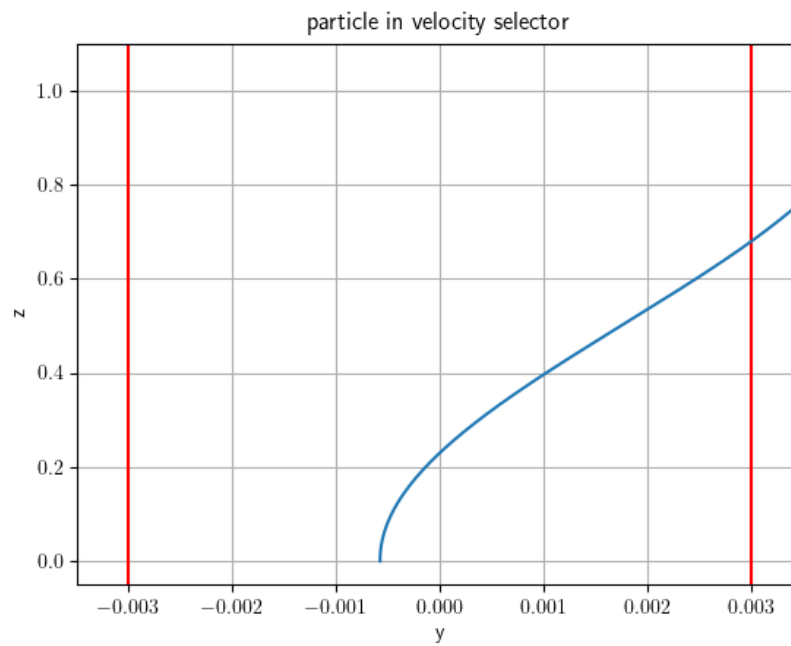
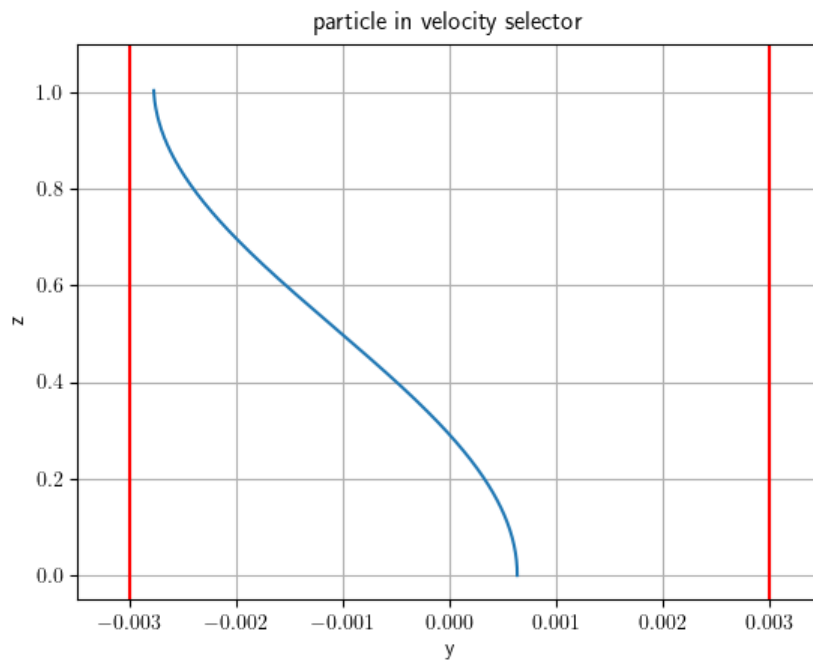
The first graph is for a particle with $E_0 = 0.8 \cdot 10^{-13}[J]$ and $y_0 = -3 \cdot 10^{-4}[m]$, we can see that the particle passes the velocity selector.



And a particle with $E_0 = 0.815 \cdot 10^{-13}[J]$ and $y_0 = 5 \cdot 10^{-4}[m]$ won't pass the velocity selector.



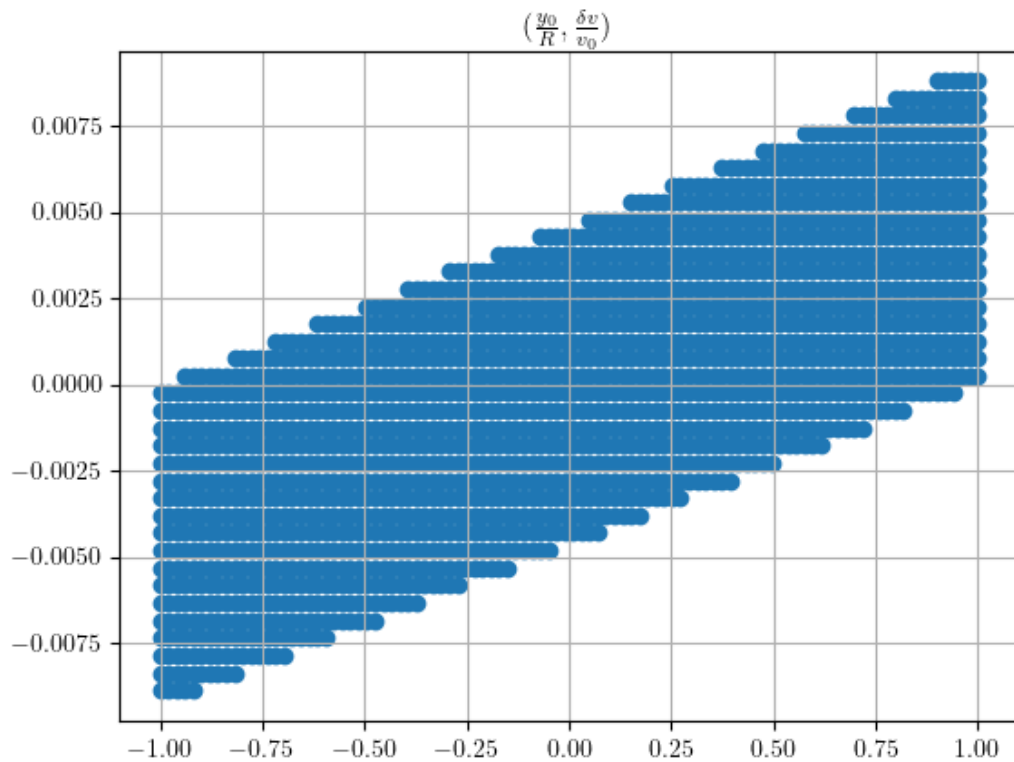
We will plot this equations in `wien_filter_numerical_integration.py` and observe the routes. The red lines are standing as the pipe boundaries and the blue line is the route of the particle.



The plane $(\frac{y_0}{R}, \frac{\delta v}{v_0})$

We will observe the plane $(\frac{y_0}{R}, \frac{\delta v}{v_0})$, and plot the dots that the particle will

pass the velocity selector. We got from the graph a parallelogram:

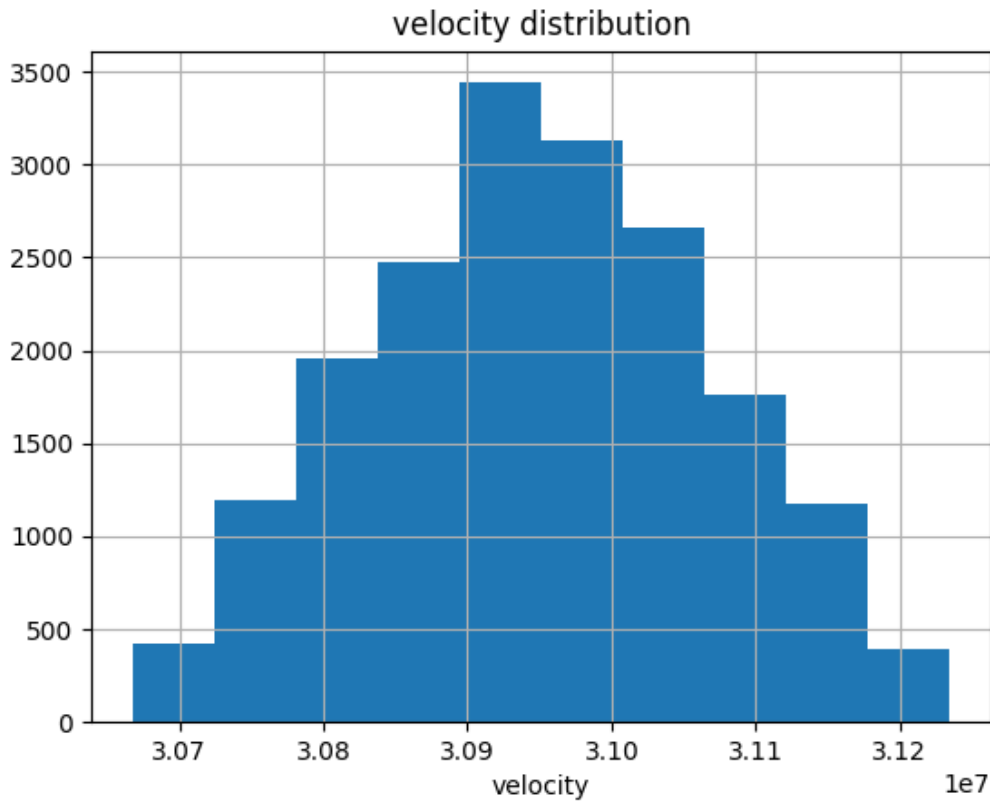


The protons beam

We will take a beam of 10^5 protons with $E_{initial} \in [E_0 - \delta E, E_0 + \delta E]$ for

$\delta E = 0.25[MeV]$ and $E_0 = 5[MeV]$ and $y_0 \in [-R, R]$ for

$R = 0.003[m]$ distributed evenly. We will calculate the distribution of the velocities of the particles that pass the velocity.



We got the following distribution of particles which makes a lot of sense.

Percent of particles which pass

We can calculate the particles that passed the velocity selector by summing the particles that have been passed, and divide by the total particles in the beam. We saw that the number of particles that passed is $n = 19,000$ and the total number of particles is $n_{tot} = 10^5$.

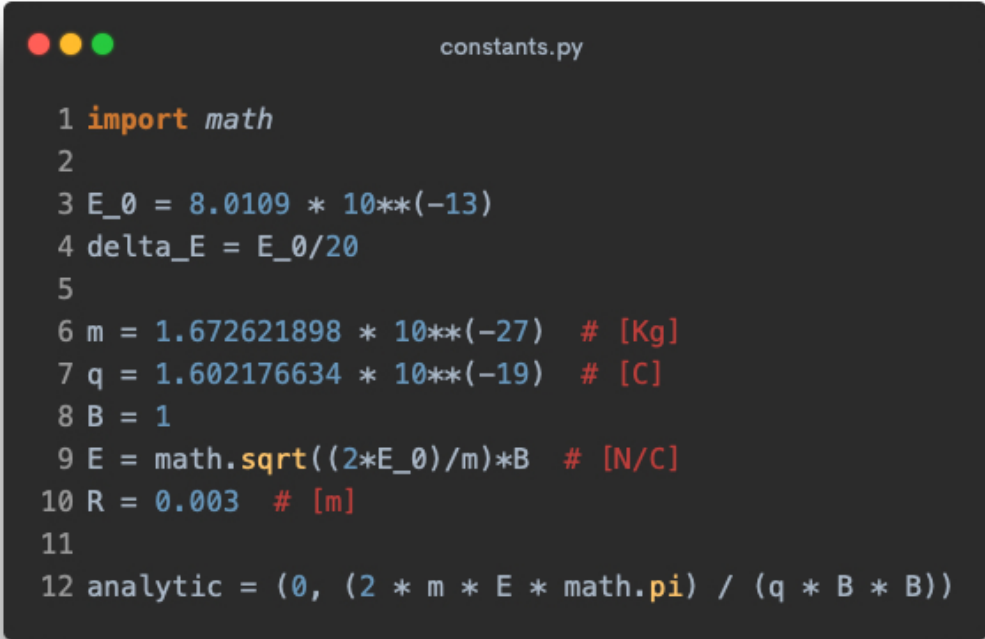
So the percent of particles which have been passed is

$$\frac{n}{n_{tot}} \cdot 100\% = \frac{19,000}{100,000} \cdot 100\% = 19\%.$$

Code Appendix

The full code can be found in the [GitHub page](#).

Link for the [desmos file](#) with the analytic solution.



```
1 import math
2
3 E_0 = 8.0109 * 10**(-13)
4 delta_E = E_0/20
5
6 m = 1.672621898 * 10**(-27) # [Kg]
7 q = 1.602176634 * 10**(-19) # [C]
8 B = 1
9 E = math.sqrt((2*E_0)/m)*B # [N/C]
10 R = 0.003 # [m]
11
12 analytic = (0, (2 * m * E * math.pi) / (q * B * B))
```



```

numerical_integration.py

1 def taylor_first_order(dt, gen_graph=False):
2     """
3          $r(t+dt) = r(t) + v(t)dt$ 
4          $v(t+dt) = v(t) + a(t)dt$ 
5     """
6     omega = (c.q * c.B) / c.m
7     T = (2 * math.pi) / omega
8     num_of_time_intervals = math.ceil(T / dt)
9
10    rz = np.zeros(num_of_time_intervals)
11    ry = np.zeros(num_of_time_intervals)
12    vz = np.zeros(num_of_time_intervals)
13    vy = np.zeros(num_of_time_intervals)
14
15    rz[0] = 0
16    ry[0] = 0
17    vz[0] = (3 * c.E) / c.B
18    vy[0] = 0
19
20    for i in range(1, num_of_time_intervals):
21        rz[i] = rz[i-1] + vz[i-1] * dt
22        ry[i] = ry[i-1] + vy[i-1] * dt
23        vz[i] = vz[i-1] + ((c.q * c.B * vy[i-1]) / c.m) * dt
24        vy[i] = vy[i-1] + ((c.q * c.E - c.q * c.B * vz[i-1]) / c.m) * dt
25
26    if gen_graph:
27        plt.rcParams['text.usetex'] = True
28
29        figure, axis = plt.subplots(2, 1, sharex=True)
30
31        axis[0].plot(ry, rz)
32        axis[0].set_title(r"${y, z}$")
33
34        axis[1].plot(vy, vz)
35        axis[1].set_title(r"${v_y, v_z}$")
36
37        plt.savefig('taylor_first_order.png')
38        plt.grid(True)
39        plt.show()
40
41    return ry[num_of_time_intervals-1], rz[num_of_time_intervals-1]

```

```

numerical_integration.py

1 def midpoint(dt, gen_graph=False):
2     omega = (c.q * c.B) / c.m
3     T = (2 * math.pi) / omega
4     num_of_time_intervals = math.ceil(T / dt)
5
6     rz = np.zeros(num_of_time_intervals)
7     ry = np.zeros(num_of_time_intervals)
8     vz = np.zeros(num_of_time_intervals)
9     vy = np.zeros(num_of_time_intervals)
10
11     rz[0] = 0
12     ry[0] = 0
13     vz[0] = (3 * c.E) / c.B
14     vy[0] = 0
15
16     def az(vy):
17         return (c.q * vy * c.B) / c.m
18
19     def ay(vz):
20         return (c.q * c.E - c.q * c.B * vz) / c.m
21
22     for i in range(1, num_of_time_intervals):
23         k1vz = az(vy[i-1]) * dt
24         k1vy = ay(vz[i - 1]) * dt
25         k2vz = az(vy[i-1] + 0.5 * k1vy) * dt
26         k2vy = ay(vz[i - 1] + 0.5 * k1vz) * dt
27
28         k1rz = vz[i - 1] * dt
29         k1ry = vy[i - 1] * dt
30         k2rz = (vz[i-1] + 0.5 * k1rz) * dt
31         k2ry = (vy[i-1] + 0.5 * k1ry) * dt
32
33         rz[i] = rz[i-1] + k2rz
34         ry[i] = ry[i - 1] + k2ry
35         vz[i] = vz[i - 1] + k2vz
36         vy[i] = vy[i - 1] + k2vy
37
38     if gen_graph:
39         plt.rcParams['text.usetex'] = True
40
41         figure, axis = plt.subplots(2, 1, sharex=True)
42
43         axis[0].plot(ry, rz)
44         axis[0].set_title(r"${y, z}$")
45
46         axis[1].plot(vy, vz)
47         axis[1].set_title(r"${v_y, v_z}$")
48
49         plt.savefig('midpoint.png')
50         plt.grid(True)
51         plt.show()
52
53     return ry[num_of_time_intervals-1], rz[num_of_time_intervals-1]

```



```
1 def runge_kutta(dt, gen_graph=False):
2     omega = (c.q * c.B) / c.m
3     T = (2 * math.pi) / omega
4     num_of_time_intervals = math.ceil(T / dt)
5
6     rz = np.zeros(num_of_time_intervals)
7     ry = np.zeros(num_of_time_intervals)
8     vz = np.zeros(num_of_time_intervals)
9     vy = np.zeros(num_of_time_intervals)
10
11     rz[0] = 0
12     ry[0] = 0
13     vz[0] = (3 * c.E) / c.B
14     vy[0] = 0
15
16     def az(vy):
17         return (c.q * vy * c.B) / c.m
18
19     def ay(vz):
20         return (c.q * c.E - c.q * c.B * vz) / c.m
21
22
23     for i in range(1, num_of_time_intervals):
24         k1vz = az(vy[i-1]) * dt
25         k1vy = ay(vz[i-1]) * dt
26         k2vz = az(vy[i-1] + 0.5 * k1vy) * dt
27         k2vy = ay(vz[i-1] + 0.5 * k1vz) * dt
28         k3vz = az(vy[i-1] + 0.5 * k2vy) * dt
29         k3vy = ay(vz[i-1] + 0.5 * k2vz) * dt
30         k4vz = az(vy[i-1] + k3vy) * dt
31         k4vy = ay(vz[i-1] + k3vz) * dt
32
33         k1rz = vz[i-1] * dt
34         k1ry = vy[i-1] * dt
35         k2rz = (vz[i-1] + 0.5 * k1rz) * dt
36         k2ry = (vy[i-1] + 0.5 * k1ry) * dt
37         k3rz = (vz[i-1] + 0.5 * k2rz) * dt
38         k3ry = (vy[i-1] + 0.5 * k2ry) * dt
39         k4rz = (vz[i-1] + k3rz) * dt
40         k4ry = (vy[i-1] + k3ry) * dt
41
42         rz[i] = rz[i-1] + (k1rz + 2 * k2rz + 2 * k3rz + k4rz) / 6
43         ry[i] = ry[i-1] + (k1ry + 2 * k2ry + 2 * k3ry + k4ry) / 6
44         vz[i] = vz[i-1] + (k1vz + 2 * k2vz + 2 * k3vz + k4vz) / 6
45         vy[i] = vy[i-1] + (k1vy + 2 * k2vy + 2 * k3vy + k4vy) / 6
46
47     if gen_graph:
48         plt.rcParams['text.usetex'] = True
49
50         figure, axis = plt.subplots(2, 1, sharex=True)
51
52         axis[0].plot(ry, rz)
53         axis[0].set_title(r"$y, z$")
54
55         axis[1].plot(vy, vz)
56         axis[1].set_title(r"$v_y, v_z$")
57
58         plt.savefig('runge_kutta.png')
59         plt.grid(True)
60         plt.show()
61
62     return ry[num_of_time_intervals-1], rz[num_of_time_intervals-1]
```

```

numerical_integration.py

1 def plot_error_graph(num_of_intervals, step):
2     times = np.zeros(num_of_intervals)
3     taylor = np.zeros(num_of_intervals)
4     mid = np.zeros(num_of_intervals)
5     runge = np.zeros(num_of_intervals)
6
7     for i in range(num_of_intervals):
8         times[i] = (i+1)*step
9         taylor[i] = error(taylor_first_order(times[i]), c.analytic)
10        mid[i] = error(midpoint(times[i]), c.analytic)
11        runge[i] = error(runge_kutta(times[i]), c.analytic)
12
13    print(times)
14    print(taylor)
15    print(mid)
16    print(runge)
17
18    plt.rcParams['text.usetex'] = True
19    plt.plot(times, taylor, label="taylor")
20    plt.plot(times, mid, label="midpoint")
21    plt.plot(times, runge, label="runge-kutta")
22
23    plt.ylabel("error")
24    plt.xlabel("dt")
25    plt.xscale("log")
26    plt.yscale("log")
27    plt.title("log log - error for dt")
28
29    plt.grid(True)
30    plt.legend()
31    plt.savefig('error.png')
32
33    plt.show()

```

```

wien_filter_numerical_integration.py

1 def runge_kutta_passes_filter(dt, E_0, y_0, gen_graph=False):
2     omega = (c.q * c.B) / c.m
3     T = (2 * math.pi) / omega
4     num_of_time_intervals = math.ceil(T / dt)
5
6     rz = np.zeros(1)
7     ry = np.zeros(1)
8     vz = np.zeros(1)
9     vy = np.zeros(1)
10
11     rz[0] = 0
12     ry[0] = y_0
13     vz[0] = math.sqrt((2 * E_0) / c.m)
14     vy[0] = 0
15
16     def az(vy):
17         return (c.q * vy * c.B) / c.m
18
19     def ay(vz):
20         return (c.q * c.E - c.q * c.B * vz) / c.m
21
22     i = 0
23     while rz[i] <= 1:
24         i += 1
25
26         k1vz = az(vy[i - 1]) * dt
27         k1vy = ay(vz[i - 1]) * dt
28         k2vz = az(vy[i - 1] + 0.5 * k1vy) * dt
29         k2vy = ay(vz[i - 1] + 0.5 * k1vz) * dt
30         k3vz = az(vy[i - 1] + 0.5 * k2vy) * dt
31         k3vy = ay(vz[i - 1] + 0.5 * k2vz) * dt
32         k4vz = az(vy[i - 1] + k3vy) * dt
33         k4vy = ay(vz[i - 1] + k3vz) * dt
34
35         k1rz = vz[i - 1] * dt
36         k1ry = vy[i - 1] * dt
37         k2rz = (vz[i - 1] + 0.5 * k1rz) * dt
38         k2ry = (vy[i - 1] + 0.5 * k1ry) * dt
39         k3rz = (vz[i - 1] + 0.5 * k2rz) * dt
40         k3ry = (vy[i - 1] + 0.5 * k2ry) * dt
41         k4rz = (vz[i - 1] + k3rz) * dt
42         k4ry = (vy[i - 1] + k3ry) * dt
43
44         rz = np.append(rz, [rz[i - 1] + (k1rz + 2 * k2rz + 2 * k3rz + k4rz) / 6])
45         ry = np.append(ry, [ry[i - 1] + (k1ry + 2 * k2ry + 2 * k3ry + k4ry) / 6])
46         vz = np.append(vz, [vz[i - 1] + (k1vz + 2 * k2vz + 2 * k3vz + k4vz) / 6])
47         vy = np.append(vy, [vy[i - 1] + (k1vy + 2 * k2vy + 2 * k3vy + k4vy) / 6])
48
49
50     if gen_graph:
51         plt.rcParams['text.usetex'] = True
52
53         plt.ylim(-0.05, 1.1)
54         plt.xlim(-0.0035, 0.0035)
55
56         plt.axvline(x=c.R, color='r', ymin= 0, ymax=1)
57         plt.axvline(x=-c.R, color='r', ymin= 0, ymax=1)
58
59         plt.plot(ry, rz)
60
61         plt.ylabel("z")
62         plt.xlabel("y")
63         plt.title("particle in velocity selector")
64         plt.grid(True)
65         plt.savefig('runge_kutta_wien_filter.png')
66         plt.show()
67
68     return -c.R < ry[i] < c.R, vz[-1]

```

```

wien_filter_numerical_integration.py

1 def error_plane():
2     energy = np.linspace(c.E_0 - c.delta_E, c.E_0 + c.delta_E, num=100)
3     radius = np.linspace(-c.R, c.R, num=100)
4
5     output_velocity = []
6     output_radius = []
7
8     for e in energy:
9         for r in radius:
10             if runge_kutta_passes_filter(10**(-10), e, r)[0]:
11                 output_velocity.append(math.sqrt(e/c.E_0)-1)
12                 output_radius.append(r/c.R)
13
14     plt.rcParams['text.usetex'] = True
15     plt.scatter(output_radius, output_velocity)
16     plt.grid(True)
17     plt.title(r"$\frac{y_0}{R}, \frac{\Delta v}{v_0}$")
18     plt.grid(True)
19     plt.savefig('error_plane.png')
20     plt.show()

```

```

wien_filter_numerical_integration.py

1 def velocity_distribution(num_of_particles):
2     energy = np.linspace(c.E_0 - c.delta_E, c.E_0 + c.delta_E, num=math.ceil(math.sqrt(num_of_particles)))
3     radius = np.linspace(-c.R, c.R, num=math.ceil(math.sqrt(num_of_particles)))
4
5     velocities = []
6     i=0
7     for e in energy:
8         for r in radius:
9             i+=1
10             passes, vel = runge_kutta_passes_filter(10 ** (-9), e, r)
11             if passes:
12                 velocities.append(vel)
13
14             print(i)
15
16     print(velocities)
17     plt.hist(velocities)
18     plt.rcParams['text.usetex'] = True
19     plt.title("velocity distribution")
20     plt.xlabel("velocity")
21     plt.grid(True)
22     plt.savefig('velocity_distribution.png')
23     plt.show()

```