

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:

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

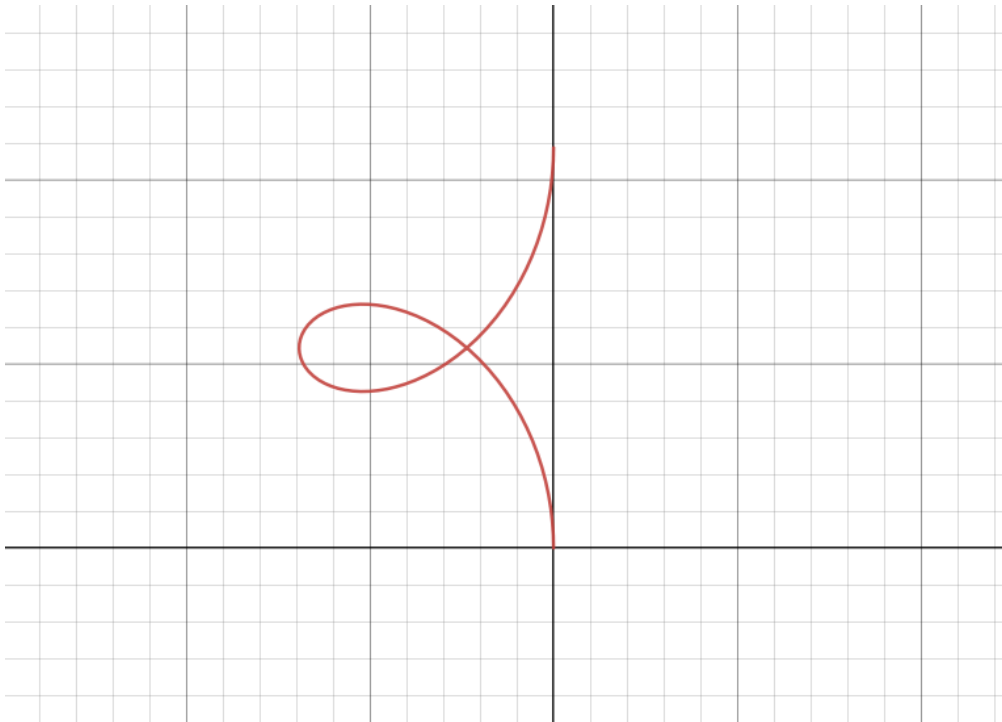
$$\Rightarrow v_y = \frac{-2E}{B}\sin\left(\frac{qB}{m}t\right)$$

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

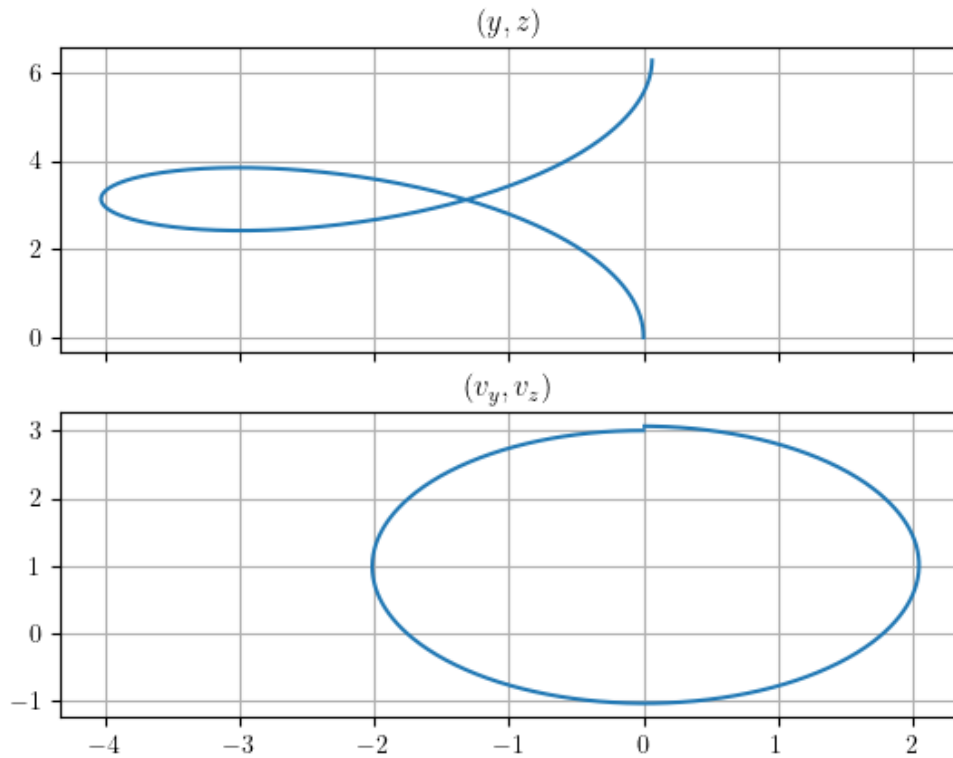
We will now solve our problem using numerical methods. Using first order Taylor approximation we will get the following relations:

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

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

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:

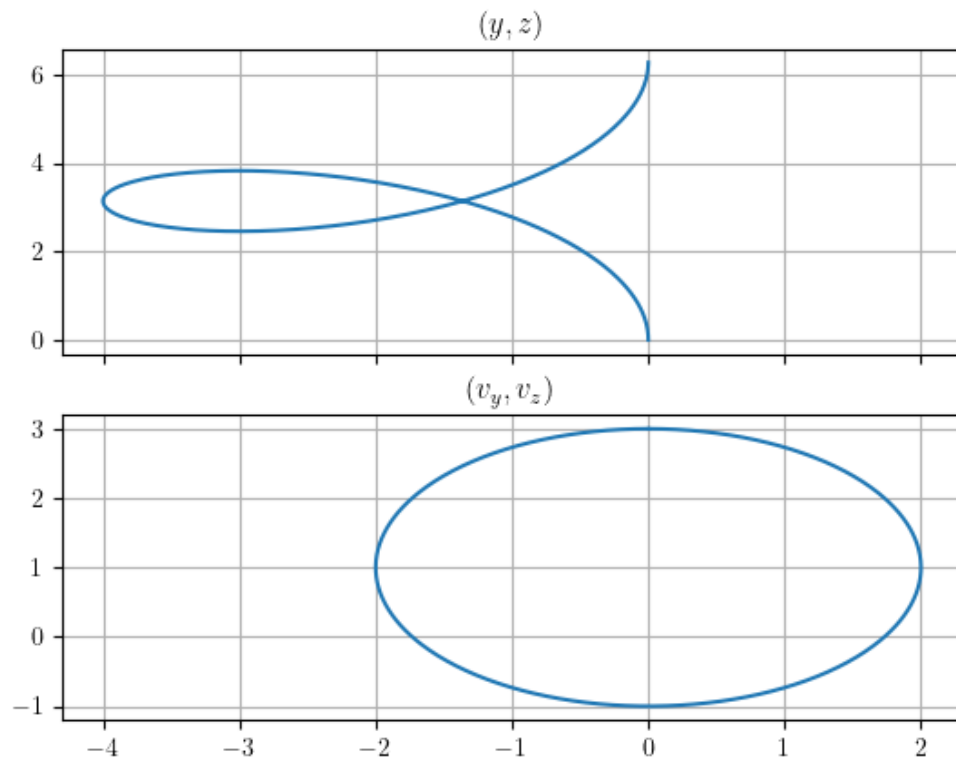
$$\begin{cases} f(t, y) = \frac{\partial f}{\partial t} \\ k_1 = \Delta t \cdot f(t, y(t)) \\ k_2 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_1}{2})) \end{cases}$$

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:

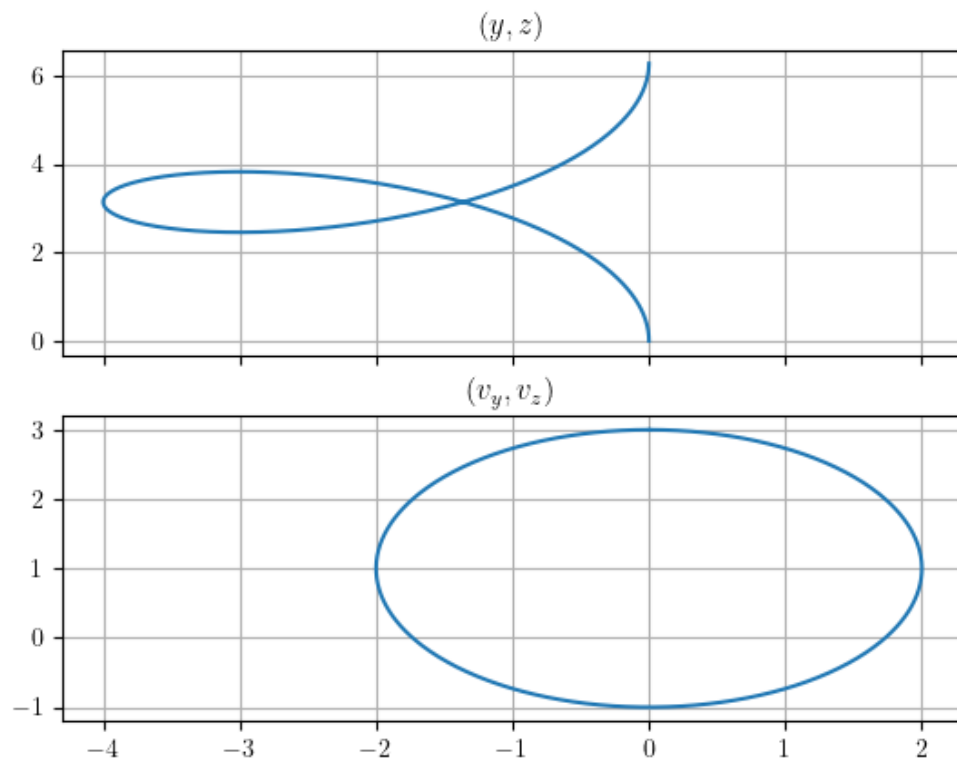
$$\begin{cases} k_1 = \Delta t \cdot f(t, y(t)) \\ k_2 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_1}{2})) \\ k_3 = \Delta t \cdot f(t + \frac{\Delta t}{2}, y(t + \frac{k_2}{2})) \\ k_4 = \Delta t \cdot f(t + \Delta t, y(t + k_3)) \end{cases}$$

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:



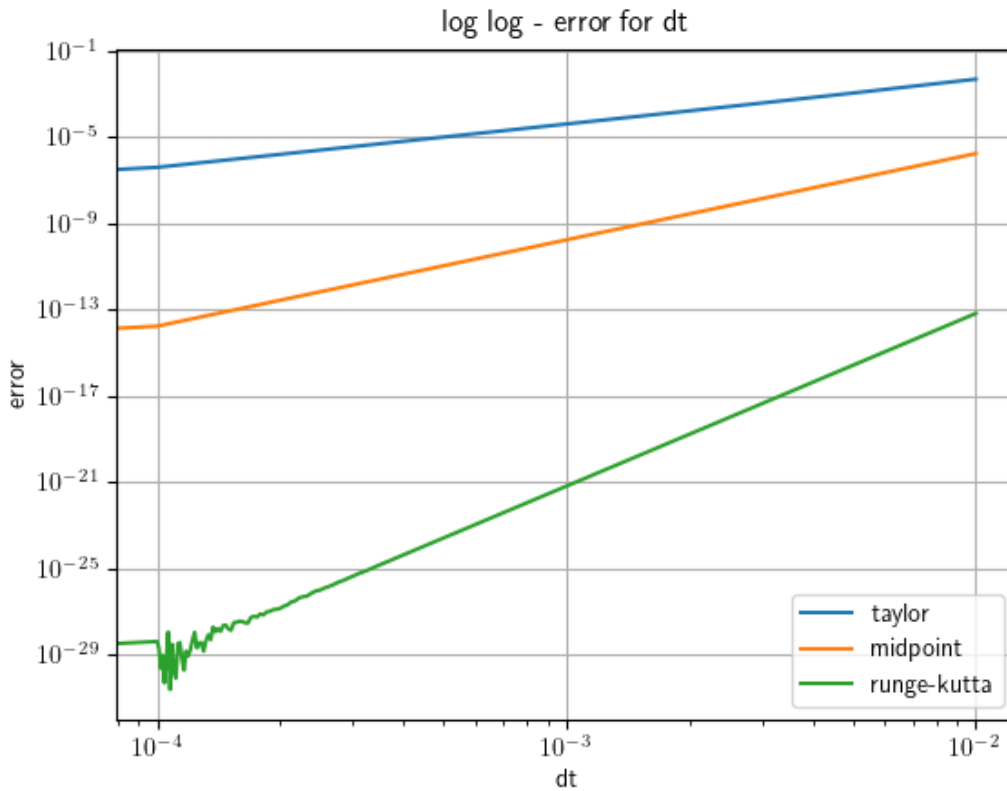
Benchmark

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

In the analytical solution we get:

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

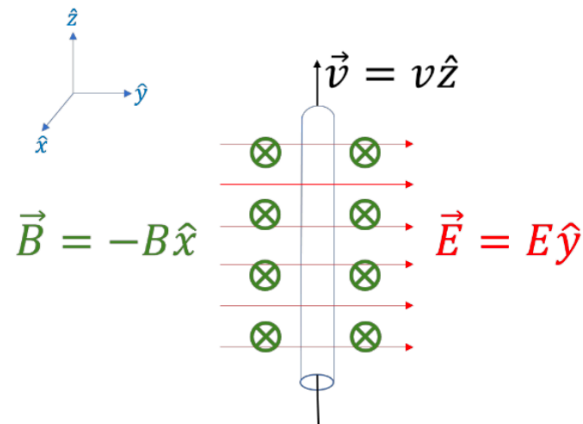
We will calculate the error (euclidian distance from the analytical to the numeric) and plot it against dt. The graph in log log scale is:



We can see the differences in the slopes of the graphs, which corresponds to the exponent of the approximation. Such a beautiful result.

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 $E_0 = 5[MeV]$ and $y_0 \in [-R, R]$ for $R = 0.003[m]$. We can derive the

velocity from the energy using $v_0 = \sqrt{\frac{2E_0}{m}}$. We will get:

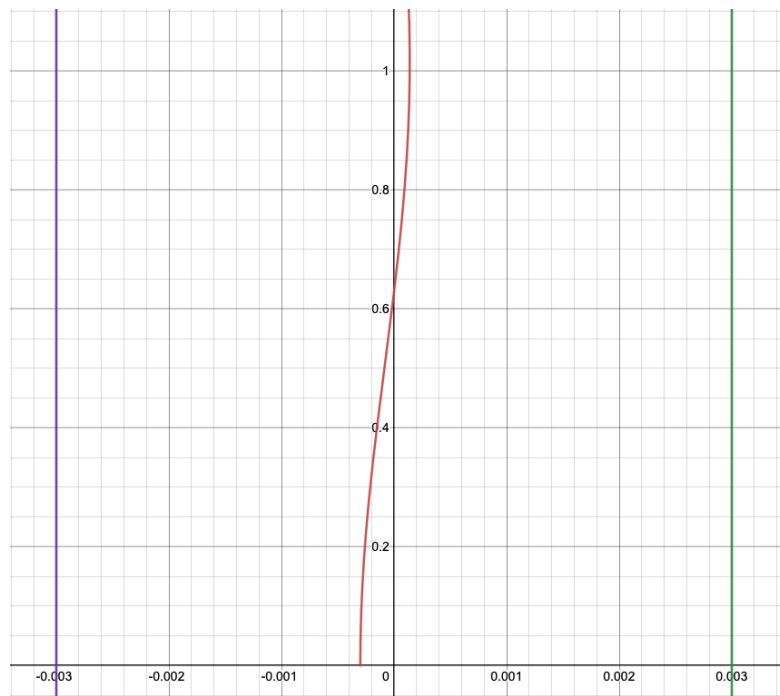
$$\begin{cases} v_z = (v_0 - \frac{E}{B})\cos(\frac{qB}{m}t) + \frac{E}{B} \\ v_y = (\frac{E}{B} - v_0)\sin(\frac{qB}{m}t) \end{cases}$$

And after integration,

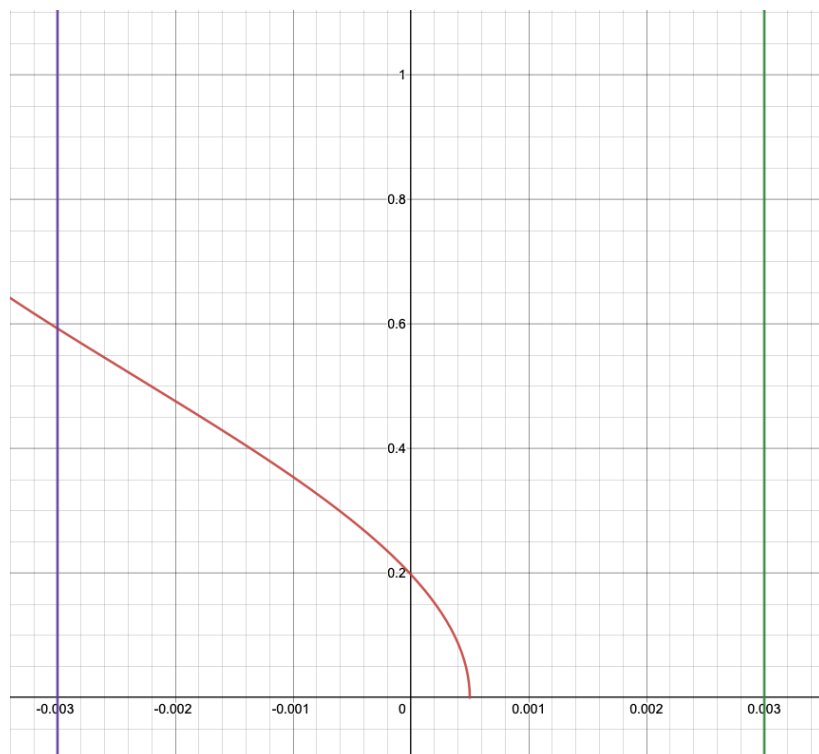
$$\begin{cases} r_z = (v_0 - \frac{E}{B})\frac{m}{qB}\sin(\frac{qB}{m}t) + \frac{E}{B}t \\ r_y = \frac{m}{qB}(v_0 - \frac{E}{B})(\cos(\frac{qB}{m}t) - 1) + y_0 \end{cases}$$

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.

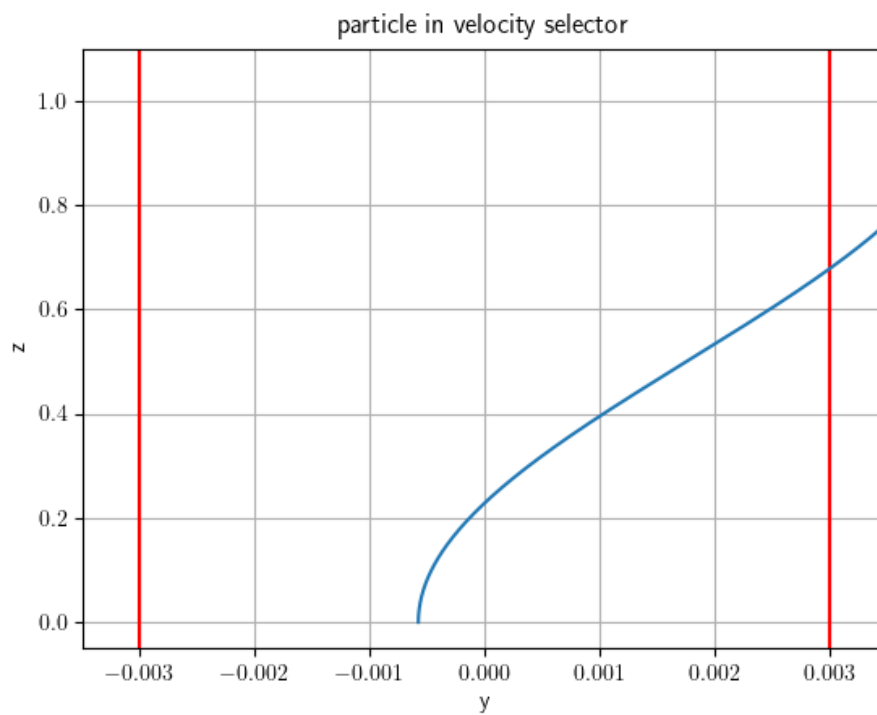
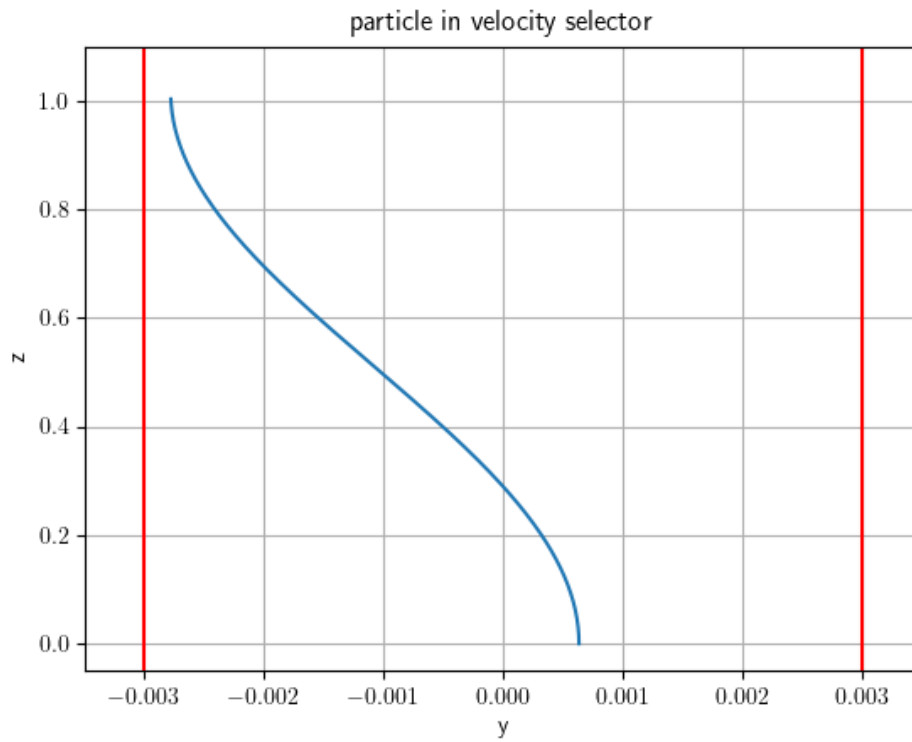
The first graph is for a particle with $E_0 = 0.8 \cdot 10^{-13}[J]$ and $y_0 = -3 \cdot 10^{-4}[m]$, we can see that the particle passes the velocity selector.



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.



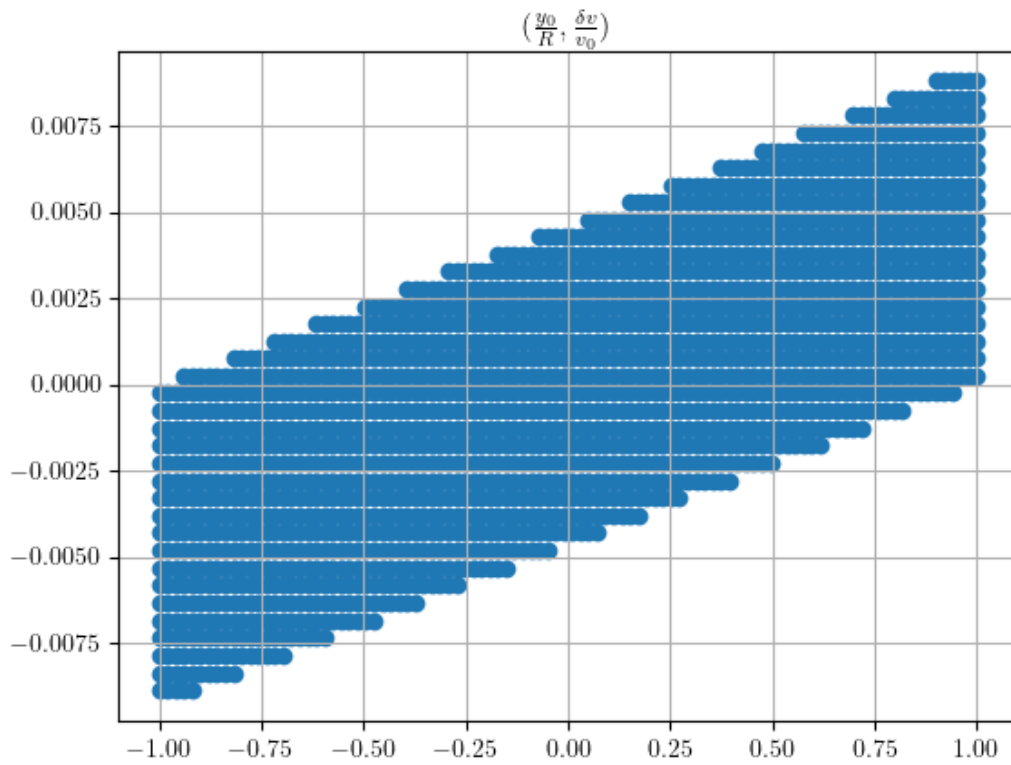
We will plot these equations in `wien_filter_numerical_integration.py` and observe the routes. The red lines are standing as the pipe boundaries and the blue line is the route of the particle.



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:

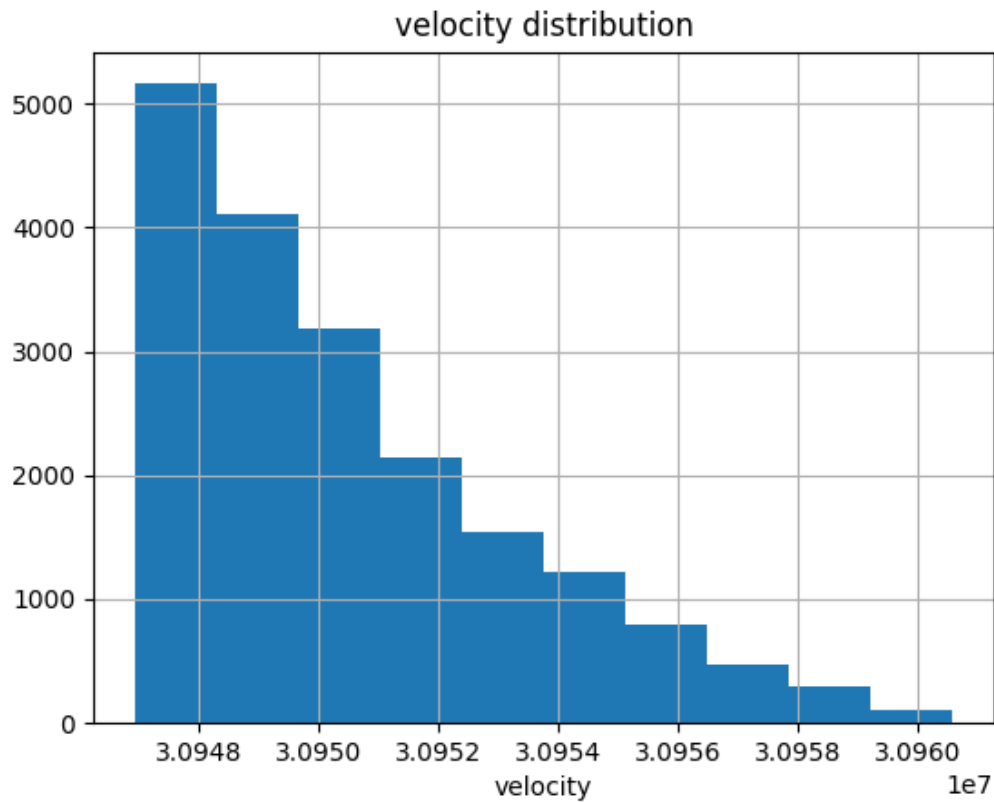


The protons beam

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 selector.

For $B = 0.5T$ we would get



We can see that the distribution is not symmetric and particles with lower speed will more tend to pass because of the force acting in the z axis.

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

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

The error can be evaluated using summing the tiny area around the parallelogram which affected by the space between 2 samples which 1 is inside the parallelogram and the other one is outside.

Can be approximated using the perimeter of the parallelogram times δl .

We can also can calculate the error by taking the variance of multiple samples. After running the calculation we would get variance of 0.42 % .

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 import numpy as np
3 import matplotlib.pyplot as plt
4
5 import constants as c
6
7
8 def taylor_first_order(num_of_time_intervals,
9   gen_graph=False):
10     """
11         
$$r(t+dt) = r(t) + v(t)dt$$

12         
$$v(t+dt) = v(t) + a(t)dt$$

13     """
14     omega = (c.q * c.B) / c.m
15     T = (2 * math.pi) / omega
16     dt = T / num_of_time_intervals
17
18     rz = np.zeros(num_of_time_intervals+1)
19     ry = np.zeros(num_of_time_intervals+1)
20     vz = np.zeros(num_of_time_intervals+1)
21     vy = np.zeros(num_of_time_intervals+1)
22
23     rz[0] = 0
24     ry[0] = 0
25     vz[0] = (3 * c.E) / c.B
26     vy[0] = 0
27
28     for i in range(1, num_of_time_intervals+1):
29         rz[i] = rz[i-1] + vz[i-1] * dt
30         ry[i] = ry[i-1] + vy[i-1] * dt
31         vz[i] = vz[i-1] + ((c.q * c.B * vy[i-1]
32         ]) / c.m) * dt
33         vy[i] = vy[i-1] + ((c.q * c.E - c.q * c.B
34         * vz[i-1]) / c.m) * dt
35
36     if gen_graph:
37         plt.rcParams['text.usetex'] = True
38
39         figure, axis = plt.subplots(2, 1, sharex=
40     True)

```

```

38         axis[0].plot(ry, rz)
39         axis[0].set_title(r"$$(y, z)$")
40
41         axis[1].plot(vy, vz)
42         axis[1].set_title(r"$$(v_y, v_z)$")
43
44         axis[0].grid(True)
45         axis[1].grid(True)
46         plt.savefig('taylor_first_order.png')
47         plt.show()
48
49     return ry[num_of_time_intervals], rz[
num_of_time_intervals]
50
51
52 def midpoint(num_of_time_intervals, gen_graph=False
):
53     omega = (c.q * c.B) / c.m
54     T = (2 * math.pi) / omega
55     dt = T / num_of_time_intervals
56
57     rz = np.zeros(num_of_time_intervals+1)
58     ry = np.zeros(num_of_time_intervals+1)
59     vz = np.zeros(num_of_time_intervals+1)
60     vy = np.zeros(num_of_time_intervals+1)
61
62     rz[0] = 0
63     ry[0] = 0
64     vz[0] = (3 * c.E) / c.B
65     vy[0] = 0
66
67     def az(vy):
68         return (c.q * vy * c.B) / c.m
69
70     def ay(vz):
71         return (c.q * c.E - c.q * c.B * vz) / c.m
72
73     for i in range(1, num_of_time_intervals+1):
74         k1vz = az(vy[i-1]) * dt
75         k1vy = ay(vz[i - 1]) * dt
76         k2vz = az(vy[i-1] + 0.5 * k1vy) * dt

```

```

77         k2vy = ay(vz[i - 1] + 0.5 * k1vz) * dt
78
79         # k1rz = vz[i - 1] * dt
80         # k1ry = vy[i - 1] * dt
81         k2rz = (vz[i-1] + 0.5 * k1vz) * dt
82         k2ry = (vy[i-1] + 0.5 * k1vy) * dt
83
84         rz[i] = rz[i-1] + k2rz
85         ry[i] = ry[i - 1] + k2ry
86         vz[i] = vz[i - 1] + k2vz
87         vy[i] = vy[i - 1] + k2vy
88
89     if gen_graph:
90         plt.rcParams['text.usetex'] = True
91
92         figure, axis = plt.subplots(2, 1, sharex=
True)
93
94         axis[0].plot(ry, rz)
95         axis[0].set_title(r"$(y, z)$")
96
97         axis[1].plot(vy, vz)
98         axis[1].set_title(r"$(v_y, v_z)$")
99
100        axis[0].grid(True)
101        axis[1].grid(True)
102        plt.savefig('midpoint.png')
103        plt.show()
104
105        return ry[num_of_time_intervals], rz[
num_of_time_intervals]
106
107
108 def runge_kutta(num_of_time_intervals, gen_graph=
False):
109     omega = (c.q * c.B) / c.m
110     T = (2 * math.pi) / omega
111     dt = T / num_of_time_intervals
112
113     rz = np.zeros(num_of_time_intervals+1)
114     ry = np.zeros(num_of_time_intervals+1)

```

```

115     vz = np.zeros(num_of_time_intervals+1)
116     vy = np.zeros(num_of_time_intervals+1)
117
118     rz[0] = 0
119     ry[0] = 0
120     vz[0] = (3 * c.E) / c.B
121     vy[0] = 0
122
123     def az(vy):
124         return (c.q * vy * c.B) / c.m
125
126     def ay(vz):
127         return (c.q * c.E - c.q * c.B * vz) / c.m
128
129
130     for i in range(1, num_of_time_intervals+1):
131         k1vz = az(vy[i-1]) * dt
132         k1vy = ay(vz[i - 1]) * dt
133         k2vz = az(vy[i-1] + 0.5 * k1vy) * dt
134         k2vy = ay(vz[i - 1] + 0.5 * k1vz) * dt
135         k3vz = az(vy[i - 1] + 0.5 * k2vy) * dt
136         k3vy = ay(vz[i - 1] + 0.5 * k2vz) * dt
137         k4vz = az(vy[i - 1] + k3vy) * dt
138         k4vy = ay(vz[i - 1] + k3vz) * dt
139
140         k1rz = vz[i - 1] * dt
141         k1ry = vy[i - 1] * dt
142         k2rz = (vz[i - 1] + 0.5 * k1vz) * dt
143         k2ry = (vy[i - 1] + 0.5 * k1vy) * dt
144         k3rz = (vz[i - 1] + 0.5 * k2vz) * dt
145         k3ry = (vy[i - 1] + 0.5 * k2vy) * dt
146         k4rz = (vz[i - 1] + k3vz) * dt
147         k4ry = (vy[i - 1] + k3vy) * dt
148
149         rz[i] = rz[i - 1] + (k1rz + 2 * k2rz + 2
150 * k3rz + k4rz) / 6
151         ry[i] = ry[i - 1] + (k1ry + 2 * k2ry + 2
152 * k3ry + k4ry) / 6
153         vz[i] = vz[i - 1] + (k1vz + 2 * k2vz + 2
154 * k3vz + k4vz) / 6
155         vy[i] = vy[i - 1] + (k1vy + 2 * k2vy + 2

```

```

152 * k3vy + k4vy) / 6
153
154     if gen_graph:
155         plt.rcParams['text.usetex'] = True
156
157         figure, axis = plt.subplots(2, 1, sharex=
True)
158
159         axis[0].plot(ry, rz)
160         axis[0].set_title(r"$(y, z)$")
161
162         axis[1].plot(vy, vz)
163         axis[1].set_title(r"$(v_y, v_z)$")
164
165         axis[0].grid(True)
166         axis[1].grid(True)
167         plt.savefig('runge_kutta.png')
168         plt.show()
169
170     return ry[num_of_time_intervals], rz[
num_of_time_intervals]
171
172
173 def error(numeric, analytic):
174     return (numeric[0] - analytic[0])**2 + (
numeric[1] - analytic[1])**2
175
176
177 def plot_error_graph():
178     omega = (c.q * c.B) / c.m
179     T = (2 * math.pi) / omega
180     times = np.zeros(101) # N number of samples,
181     taylor = np.zeros(101)
182     mid = np.zeros(101)
183     runge = np.zeros(101)
184     i=0
185     for n in np.linspace(100, 10000, 100):
186         print(n)
187
188         times[i] = T/n
189         taylor[i] = error(taylor_first_order(int(n

```

```
189 )) , c.analytic)
190         mid[i] = error(midpoint(int(n)), c.
    analytic)
191         runge[i] = error(runge_kutta(int(n)), c.
    analytic)
192         i += 1
193
194     print(times)
195     print(taylor)
196     print(mid)
197     print(runge)
198
199     plt.rcParams['text.usetex'] = True
200     plt.plot(times, taylor, label="taylor")
201     plt.plot(times, mid, label="midpoint")
202     plt.plot(times, runge, label="runge-kutta")
203
204     plt.ylabel("error")
205     plt.xlabel("dt")
206     plt.xscale("log")
207     plt.yscale("log")
208     plt.title("log log - error for dt")
209
210     plt.grid(True)
211     plt.legend()
212     plt.savefig('error.png')
213
214     plt.show()
215
216
217
218
```

```

1 import math
2 import random
3
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 import constants
8 import constants as c
9
10
11 def runge_kutta_passes_filter(dt, E_0, y_0,
    gen_graph=False):
12     omega = (c.q * c.B) / c.m
13     T = (2 * math.pi) / omega
14     num_of_time_intervals = math.ceil(T / dt)
15
16     rz = np.zeros(1)
17     ry = np.zeros(1)
18     vz = np.zeros(1)
19     vy = np.zeros(1)
20
21     rz[0] = 0
22     ry[0] = y_0
23     vz[0] = math.sqrt((2 * E_0)/c.m)
24     vy[0] = 0
25
26     def az(vy):
27         return (c.q * vy * c.B) / c.m
28
29     def ay(vz):
30         return (c.q * c.E - c.q * c.B * vz) / c.m
31
32     i = 0
33     while rz[i] <= 1:
34         i += 1
35
36         k1vz = az(vy[i - 1]) * dt
37         k1vy = ay(vz[i - 1]) * dt
38         k2vz = az(vy[i - 1] + 0.5 * k1vy) * dt
39         k2vy = ay(vz[i - 1] + 0.5 * k1vz) * dt
40         k3vz = az(vy[i - 1] + 0.5 * k2vy) * dt

```

```

41         k3vy = ay(vz[i - 1] + 0.5 * k2vz) * dt
42         k4vz = az(vy[i - 1] + k3vy) * dt
43         k4vy = ay(vz[i - 1] + k3vz) * dt
44
45         k1rz = vz[i - 1] * dt
46         k1ry = vy[i - 1] * dt
47         k2rz = (vz[i - 1] + 0.5 * k1vz) * dt
48         k2ry = (vy[i - 1] + 0.5 * k1vy) * dt
49         k3rz = (vz[i - 1] + 0.5 * k2vz) * dt
50         k3ry = (vy[i - 1] + 0.5 * k2vy) * dt
51         k4rz = (vz[i - 1] + k3vz) * dt
52         k4ry = (vy[i - 1] + k3vy) * dt
53
54         rz = np.append(rz, [rz[i - 1] + (k1rz + 2
55             * k2rz + 2 * k3rz + k4rz) / 6])
56         ry = np.append(ry, ry[i - 1] + (k1ry + 2 *
57             k2ry + 2 * k3ry + k4ry) / 6)
58         vz = np.append(vz, vz[i - 1] + (k1vz + 2 *
59             k2vz + 2 * k3vz + k4vz) / 6)
60         vy = np.append(vy, vy[i - 1] + (k1vy + 2 *
61             k2vy + 2 * k3vy + k4vy) / 6)
62
63         if gen_graph:
64             plt.rcParams['text.usetex'] = True
65
66             plt.ylim(-0.05, 1.1)
67             plt.xlim(-0.0035, 0.0035)
68
69             plt.axvline(x=c.R, color='r', ymin= 0, ymax
70 =1)
71             plt.axvline(x=-c.R, color='r', ymin= 0,
72 ymax=1)
73
74             plt.plot(ry, rz)
75
76             plt.ylabel("z")
77             plt.xlabel("y")
78             plt.title("particle in velocity selector")
79             plt.grid(True)
80             plt.savefig('runge_kutta_wien_filter.png')

```



```

76         plt.show()
77
78     return -c.R < ry[i] < c.R, math.sqrt(vz[-1]**2
+ vy[-1]**2)
79
80
81
82 def error_plane():
83     energy = np.linspace(c.E_0 - c.delta_E, c.E_0
+ c.delta_E, num=100)
84     radius = np.linspace(-c.R, c.R, num=100)
85
86     output_velocity = []
87     output_radius = []
88
89     count = 0
90
91     for e in energy:
92         for r in radius:
93             if runge_kutta_passes_filter(10**(-11
), e, r,)[0]:
94                 count += 1
95                 output_velocity.append(math.sqrt(e
/c.E_0)-1)
96                 output_radius.append(r/c.R)
97
98     plt.rcParams['text.usetex'] = True
99     plt.scatter(output_radius, output_velocity)
100     plt.grid(True)
101     plt.title(r"$\left(\frac{y_0}{R}, \frac{\Delta v}{v_0}\right)$")
102     plt.grid(True)
103     plt.savefig('error_plane.png')
104     plt.show()
105
106
107 def velocity_distribution(num_of_particles):
108     energy = np.linspace(c.E_0 - c.delta_E, c.E_0
+ c.delta_E, num=math.ceil(math.sqrt(
num_of_particles)))
109     radius = np.linspace(-c.R, c.R, num=math.ceil(

```

```

109 math.sqrt(num_of_particles)))
110
111     velocities = []
112     i=0
113     for e in energy:
114         for r in radius:
115             i+=1
116             passes, vel =
runge_kutta_passes_filter(10 ** (-9), e, r)
117             if passes:
118                 velocities.append(vel)
119
120
121     plt.hist(velocities, bins=100)
122     plt.rcParams['text.usetex'] = True
123     plt.title(r"velocity distribution")
124     plt.xlabel("velocity")
125     plt.grid(True)
126     plt.savefig('velocity_distribution.png')
127     plt.show()
128
129
130 def calculate_area():
131     output_velocity = []
132     output_radius = []
133
134     count = 0
135
136     for i in range(10**4):
137         e = c.E_0 + 2 * c.delta_E * (random.random
138         (-0.5)
139         r = -c.R + 2 * c.R * random.random()
140         if runge_kutta_passes_filter(10 ** (-10),
141         e, r, )[0]:
142             count += 1
143             output_velocity.append(math.sqrt(e / c
144             .E_0) - 1)
145             output_radius.append(r / c.R)
146
147     return (count/10**4) * 100 # return pass
percentage

```

```
145
146
147 def calculate_variance():
148     arr = list()
149     for i in range(25):
150         res = calculate_area()
151         arr.append(res)
152
153     print("The variance is:", np.std(arr))
154
155
156
157
```