

## 0.1 Intermediate Versions

### 0.1.1 $L_{out}$ Optimization For Single $e^-$

Initial step of improving *POC* towards *RhodotronSimulation* was to implement  $L_{out}$  optimizations to help optimazing magnet designs, as discussed in *section ??*. First approach was to hang the  $e^-$  outside of the cavity for  $t_{out} = L_{out}/v$ , then inject it back to the cavity with reversed  $\vec{v}$ . Then sweep the  $t_{out}$  parameter to find the optimal value. This simple implementation can be found in *figure A.1* of *Appendix A*.

Although the results from this optimization sweep were promising after they were simulated with *CST*, simulating one particle would not be sufficiently useful for designing a magnet.

### 0.1.2 $\phi_{lag}$ Optimization For Bunches

After successfully accelerating single  $e^-$ , particle bunches were implemented to approximate a real  $e^-$  gun. They were modeled as  $N$  electrons fired from an  $e^-$  gun at even time intervals. This approach was taken because the amount of time gun fires, defined as *Gun Active Time*,  $t_g$ , is a crucial part of pulsing  $e^-$  gun design.

Addition of bunches would immediately proven useful when finding optimal gun phase lag.  $\phi_{lag}$  for a bunch was defined as the RF phase when the first  $e^-$  of the bunch entered the cavity, it defines the starting time of the current pass. To use the *parameter sweep* method, as used in  $L_{out}$  optimization, relevant bunch characteristics are defined as follows:

- $\mu E$ : Average energy
- $E_{rms}$ : Root mean square of energy
- $R_{rms}$ : Root mean square of  $e^-$  positions

Optimal  $\phi_{lag}$  would produce maximum  $\mu E$  while minimizing  $E_{rms}$  &  $R_{rms}$ . For the first pass,  $E_{rms}$  and  $R_{rms}$  would be vaguely dependent of each other; therefore, early implementation of  $\phi_{lag}$  sweep was based on minimizing  $E_{rms}$  during simulation (*see figure 1*). For  $\mu E$  considerations, data from the software would be analyzed either manually or by using external tools such as *ROOT*.

---

```

1  int phase_opt(int phase_sweep_range){
2      double minrms = 1;
3      int opt_phase;
4      for(int RFphase = -phase_sweep_range; RFphase <= phase_sweep_range; RFphase++){
5          Bunch bunch1(RFphase);
6          double t1 = 0;
7          bunch1.bunch_gecis_t(t1);
8          bunch1.reset_pos();
9
10         if( bunch1.E_rms() < minrms ){
11             minrms = bunch1.E_rms();
12             opt_phase = RFphase;
13         }
14     }
15     return opt_phase;
16 }

```

---

Figure 1:  $\phi_{lag}$  Optimization For Initial Bunch Design

Since  $\phi_{lag}$  is relatively easy to change after production, another version of *figure 1* that was modified for given magnet design parameters was also implemented (see *figure A.2 in Appendix A*). This version can be useful for optimizing  $\phi_{lag}$  in case of production issues in magnets.

After the bunch and  $\phi_{lag}$  sweep implementations,  $L_{out}$  sweep was also updated to minimize  $E_{rms}$ .  $\rho$  and  $L$  calculations, using *equations ?? and ??*, were also integrated. Two example runs can be found in *figures B.1 and B.2 of Appendix B*.

### 0.1.3 Simulation in 3D

After successfully implementing  $\mathbf{e}^- - \vec{\mathbf{E}}$  interaction in 1D and confirming the usefulness of this tool, the decision was made to proceed with implementing a 3D version of the *Rhodotron Simulation*. Although complete refactoring of the software was necessary, this upgrade was crucial for implementation of  $\mathbf{e}^- - \vec{\mathbf{B}}$  interaction. The refactoring effort included proper implementation of *OOP*, details of which can be seen in *Appendix A*.

Magnets were modeled as major segments of a circle, defined with  $\vec{\mathbf{r}}_{mag}$ ,  $\mathbf{R}$  and  $|\mathbf{B}|$ . For the initial implementation,  $\vec{\mathbf{B}}$  assumed to be uniform and has no leaks outside the magnet boundary (See *figure A.4 in Appendix A*).

Interaction logic for  $\mathbf{e}^- - \vec{\mathbf{E}}$  and  $\mathbf{e}^- - \vec{\mathbf{B}}$  in 3D can be found in *figure 2*.

---

```

1 vector3d CoaxialRFField::actOn(Electron& e){
2     vector3d Efield = getField(e.pos);                // Calculate E vector
3     vector3d F_m = Efield*1E6*eQMratio;              // Calculate F/m vector
4     vector3d acc = (F_m - e.vel*(e.vel*F_m)/(c*c))/e.gamma(); // Calculate a vector
5     return acc;
6 }

```

---

```

1 vector3d MagneticField::actOn(Electron& e){
2     if (isInside(e.pos) == -1)
3         return vector3d(0,0,0);
4     vector3d Bfield = getField(e.pos);                // Calculate B vector
5     vector3d F_m = (e.vel % Bfield)*eQMratio;        // Calculate F/m vector
6     vector3d acc = (F_m)/e.gamma();                  // Calculate a vector
7     return acc;
8 }

```

---

Figure 2:  $\mathbf{e}^- - \mathbf{EM}$  interaction logic from *equation ??*

Where  $*$  and  $\%$  are, dot-product and cross-product respectably. (See *figure A.3 of Appendix A*)

Simulating in 3D had one other benefit; it was now possible to visualize the results by rendering the interaction data. For this purpose, *gnuplot* was integrated into *Rhodotron Simulation* to produce 2D visualization of acceleration plane. Rendered results could be stored as *gif* animations. Two of such renders can be seen in *figure 3*.

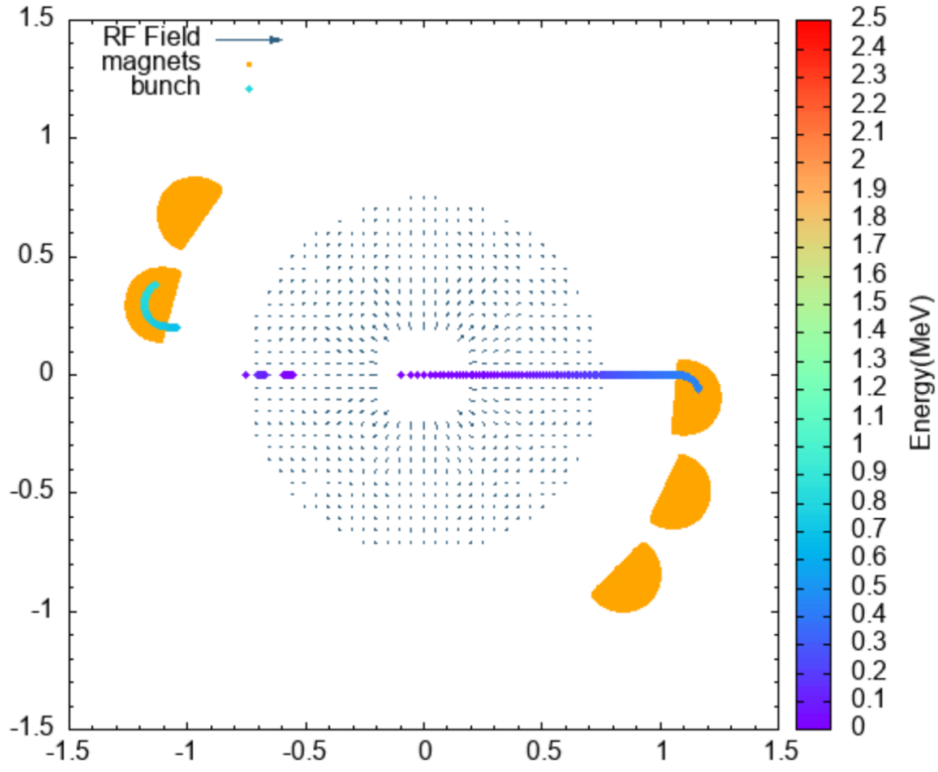
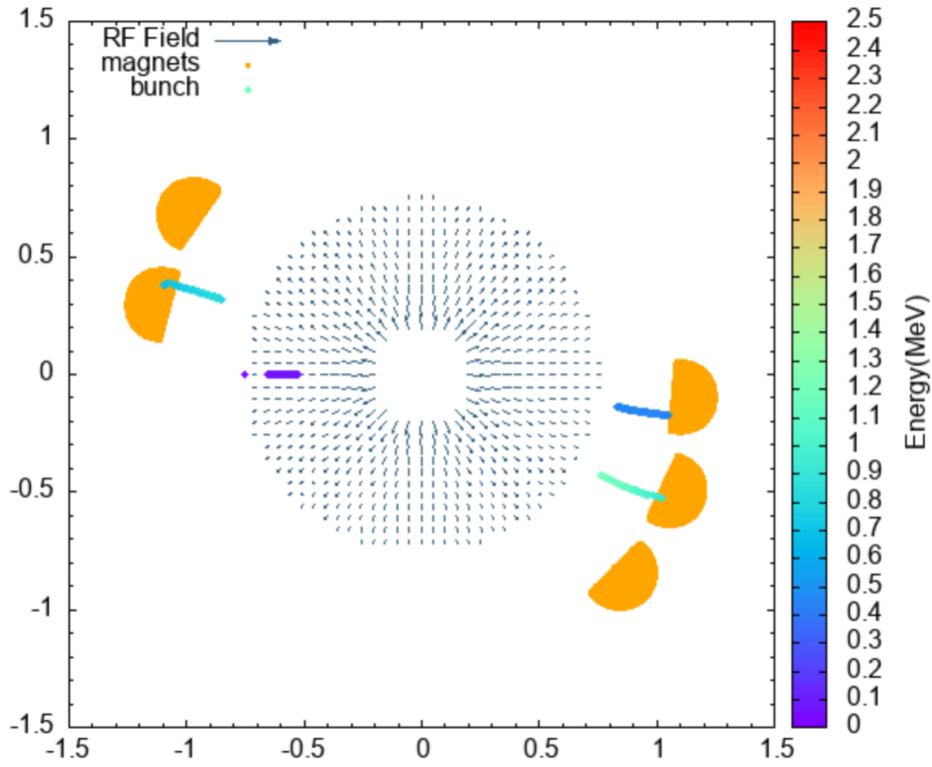
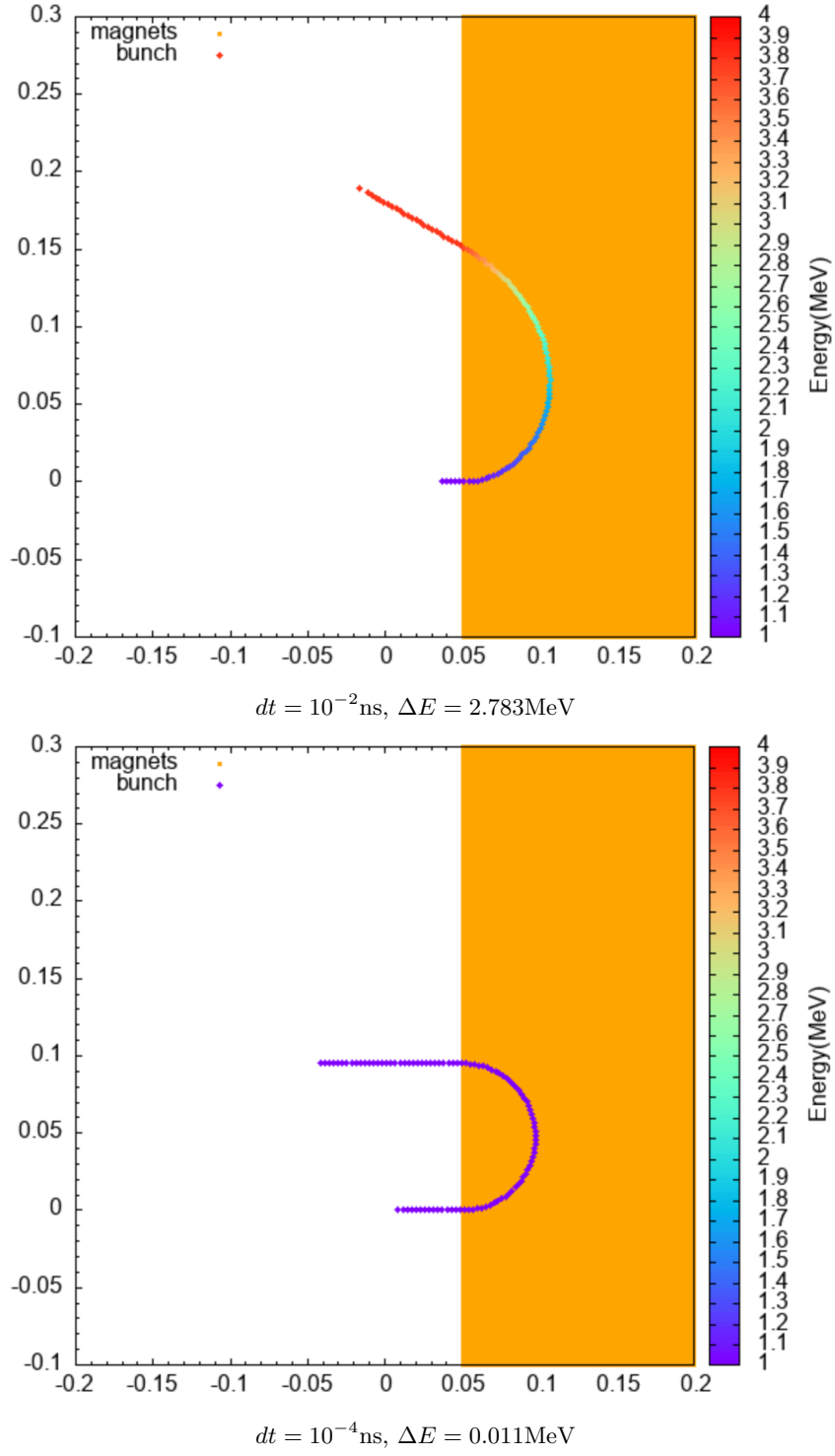
Unsynchronized  $e^-$  gun period,  $T_g = 5ns$ Synchronized  $e^-$  gun period,  $T_g = 9.3ns$ 

Figure 3: Example *gnuplot* renders of *Rhodotron Simulation*  
 $P = 12kW$ ,  $f = 107.5MHz$

#### 0.1.4 Acceleration in Magnetic Field

An issue regarding the  $\mathbf{e}^-$  -  $\vec{\mathbf{B}}$  interaction became apparent when energy gain during these interactions was observed. A setup simulation was implemented in which a bunch of  $100\mathbf{e}^-$  at 1MeV was fired to a uniform magnetic field of 0.1T placed in  $x > 0.05\text{m}$ . Initial results at  $dt = 0.01\text{ns}$  proved the suspicion of  $\mathbf{e}^-$  -  $\vec{\mathbf{B}}$  interaction being broken. However, the energy gain would decrease tremendously as  $dt$  decreased.

Figure 4: Energy gain of 1MeV bunch in  $B=0.1T$

Decreasing  $dt$  would be the best way to increase accuracy of the results; however, this is not sustainable because time and computing power. Until this point, *Rhodotron Simulation* have been using *section ??* for  $\mathbf{e}^-$  - **EM** interactions. To test newer approaches, two additional version of  $\mathbf{e}^-$  - **EM** interaction that are using *section ??* were added.

### RK4-1

First approach for integrating *section ??* into  $\mathbf{e}^-$  - **EM** interaction was to calculate  $\vec{\mathbf{a}}_E$  and  $\vec{\mathbf{a}}_B$  from *equation ??* using *RK4*. After  $\vec{\mathbf{a}}_{EM} = \vec{\mathbf{a}}_E + \vec{\mathbf{a}}_B$  was calculated,  $\mathbf{e}^-$  would move and accelerate using the *Leap-frog* method. The idea was to produce more refined interaction results, leading to improved accuracy especially in  $\mathbf{e}^-$  -  $\vec{\mathbf{B}}$ . RF field was kept static during the RK4 computation, due to ongoing *multithreading* implementation efforts. The implementation can be found in *figures A.5 and A.6*.

### RK4-2

Following the implementation of *RK4-1*, revisions were made to the integration method for *RK4* to replace *Leap-frog*. Instead of calculating  $\vec{\mathbf{a}}_{EM}$  using *RK4*,  $\vec{\mathbf{r}}$  and  $\vec{\mathbf{v}}$  would be determined directly.

These three methods were then tested in the same setup as *figure 4*. The results can be found in *figure C.1* of *Appendix C*. *RK4-1* was decided to be abandoned as it produced the same accuracy in twice the simulation time of *RK4-2*.

More rigorous testing was done with *Leap-frog* and *RK4-2* however. Still using the setup in *figure 4*, each  $dt$  configuration was simulated 10 times, calculating average and standard deviations afterwards. Results from these tests can be observed in *figure 5*.

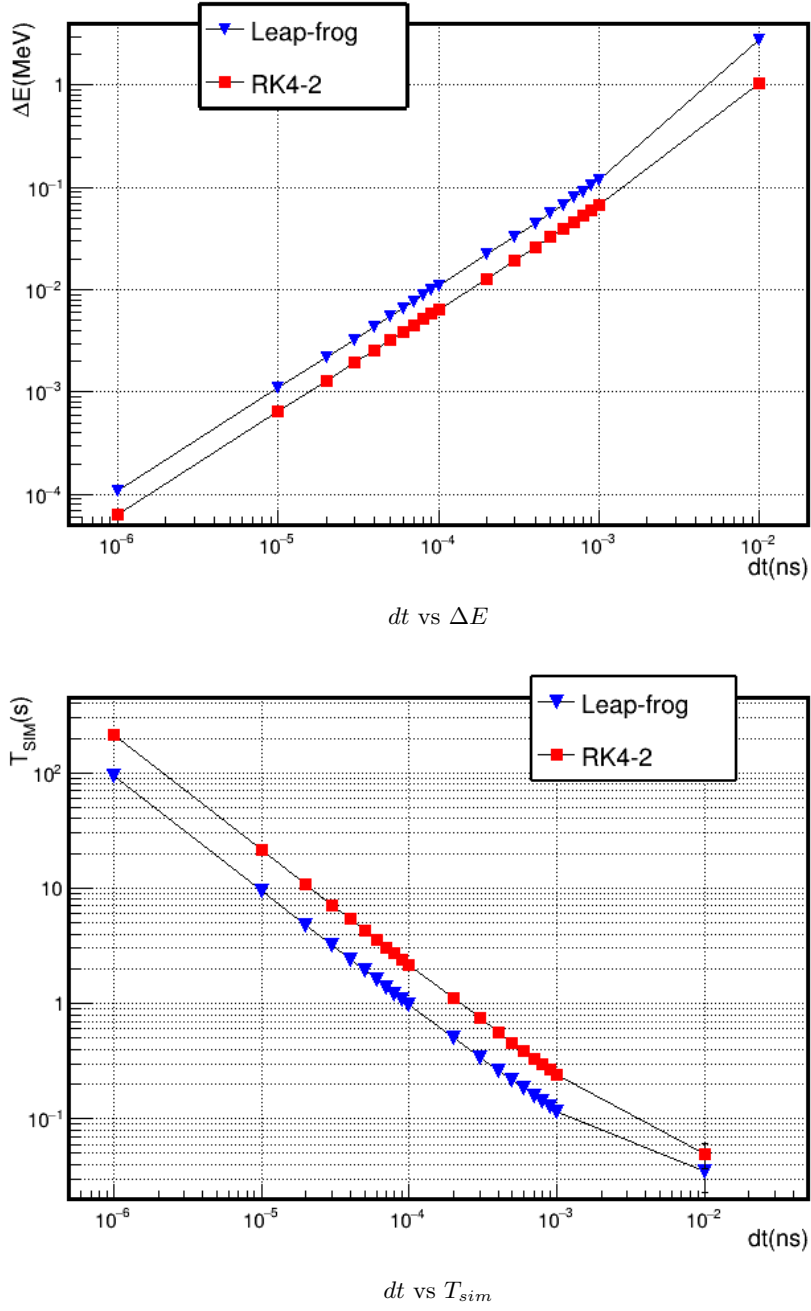


Figure 5: Comparing Leap-frog, RK4-2 performance on  $e^- - \vec{B}$  interaction  
 $E_{in} = 1\text{MeV}$ ,  $\mathbf{B}=0.1\text{T}$ ,  $t_{end} = 5\text{ns}$

The data from these tests can be found in *Tables C.1 and C.2 of Appendix C*. To investigate the

data further, one can define a performance measurement,  $F$ .

$$\begin{aligned} F &\propto 1/T \\ F &\propto 1/\Delta E \end{aligned}$$

When  $dt = 10^{-5}\text{ns}$  was taken as reference point due to providing a good balance of accuracy and computational intensity,

$$\begin{aligned} \Delta E_{LF} &= 110 \times 10^{-5} \text{MeV} \\ \Delta E_{RK}^1 &= 64 \times 10^{-5} \text{MeV} \\ T_{LF}^1 &= 9.44 \pm 0.03 \text{s} \\ T_{RK}^1 &= 21.33 \pm 0.02 \text{s} \\ \Delta E_{LF} \times T_{LF} &= 104 \times 10^{-4} \pm 10^{-4} \text{MeVs} \\ \Delta E_{RK} \times T_{RK} &= 137 \times 10^{-4} \pm 2 \times 10^{-4} \text{MeVs} \\ F_{LF}/F_{RK} &= \frac{\Delta E_{RK} \times T_{RK}}{\Delta E_{LF} \times T_{LF}^1} = 1.32 \pm 0.01 \end{aligned} \tag{1}$$

Also, observing from the data,

$$\begin{aligned} \Delta E_{LF}(dt = 3 \times 10^{-5}) &\approx \Delta E_{RK}(dt = 5 \times 10^{-5}) \approx 32.5 \times 10^{-4} \text{MeV} \\ T_{LF}(dt = 3 \times 10^{-5}) &= 3.18 \pm 0.02 \text{s} \\ T_{RK}(dt = 5 \times 10^{-5}) &= 4.30 \pm 0.01 \text{s} \\ F_{LF}/F_{RK} &= \frac{T_{RK}(dt = 5 \times 10^{-5})}{T_{LF}(dt = 3 \times 10^{-5})} = 1.4 \pm 0.1 \end{aligned} \tag{2}$$

Uncertainty of *equation 2* was taken high due to the approximation. After combining *equations 1 and 2*,  $F$  can be calculated as

$$F_{LF}/F_{RK} = 1.36 \pm 0.05 \tag{3}$$

Therefore, *Leap-frog* was found to be the better choice as it provided with  $1.36 \pm 0.05$  times accuracy in  $\mathbf{e}^- - \vec{\mathbf{B}}$  interactions at a given time with respect to *RK4*. However, *RK4* was promising in situations where decreasing the stepsize,  $dt$ , is not viable. Therefore both were integrated into *Rhodotron Simulation* for the user to decide. *RK4-2* renders from these test can be found in *figure B.3 of Appendix B*.

### 0.1.5 Acceleration in Electric Field

After the accuracy concerns regarding the  $\mathbf{e}^- - \vec{\mathbf{B}}$  interaction were raised, it was decided to test  $\mathbf{e}^- - \vec{\mathbf{E}}$  and compare the performance of *Leap-frog* and *RK4*.

As test setups, two simulation configurations were made. They aimed to test the accuracy of acceleration of a beam in parallel and perpendicular static uniform electric fields. Both configurations had an  $\mathbf{e}^-$ -gun located at  $(-0.753, 0, 0)\text{m}$ , directed at  $(1, 0, 0)$  firing electrons with the kinetic energy of 1MeV.



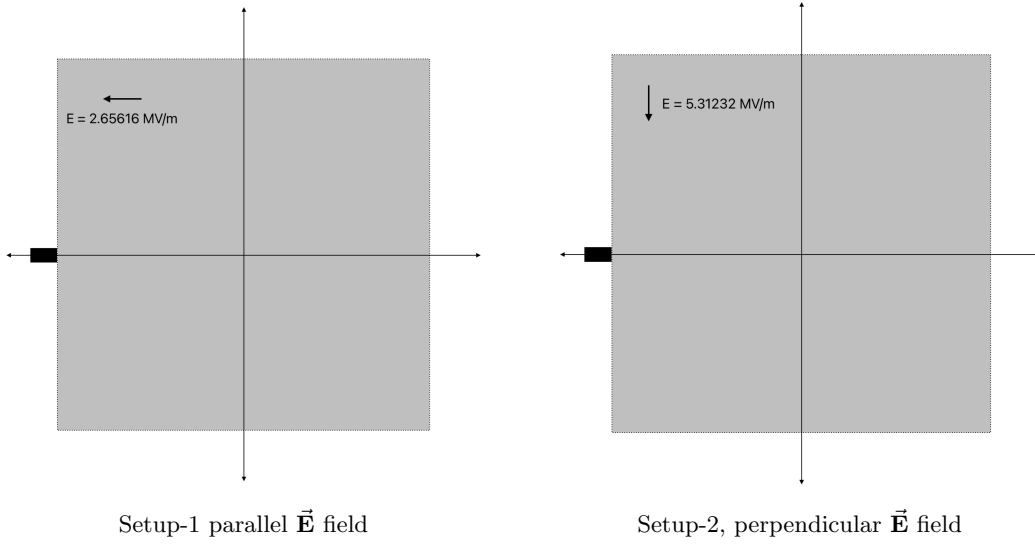


Figure 6: Illustration of test setups.

In the first test, the beam would be injected into a static uniform electric field  $\vec{E} = (-2.65616, 0, 0)$  MV/m where  $-0.753 < x < 0.753$  and  $-0.753 < y < 0.753$ ,  $\vec{E} = 0$  elsewhere.

Considering the  $\vec{E}$  is parallel to the beam path, potential difference  $V$  in the trajectory until  $(-0.753, 0, 0)$ m is

$$\Delta V^1 = - \int \vec{E} \cdot d\vec{s} \quad (4)$$

$$= - \int_{-0.753}^{0.753} -2.65616 \times dx \quad (5)$$

$$= 2.65616 \times 1.506 \quad (6)$$

$$\Delta V^1 = 4MV \quad (7)$$

$$\Delta E^1 = 4MeV \quad (8)$$

$$E_{exitTH}^1 = 5MeV \quad (9)$$

In the second test on the other hand, the beam would be injected into a different static uniform electric field,

$\vec{E} = (0, -5.31232, 0)$  MV/m where  $-0.753 < x < 0.753$  and  $-0.753 < y < 0.753$ ,  $\vec{E} = 0$  elsewhere.

$$\Delta V^2 = - \int \vec{E} \cdot d\vec{s} \quad (10)$$

$$= - \int_0^{0.753} -5.31232 \times dy \quad (11)$$

$$= 5.31232 \times 0.753 \quad (12)$$

$$\Delta V^2 = 4MV \quad (13)$$

$$\Delta E^2 = 4MeV \quad (14)$$

$$E_{exitTH}^2 = 5MeV \quad (15)$$

Therefore, in the both tests, the beam was expected to exit  $\vec{\mathbf{E}}$  with  $E_{exitTH} = 5$  MeV. To also measure the variance in simulation completion times,  $T_{sim}$ , set of 10 runs were completed at the configuration for each  $dt$  value.

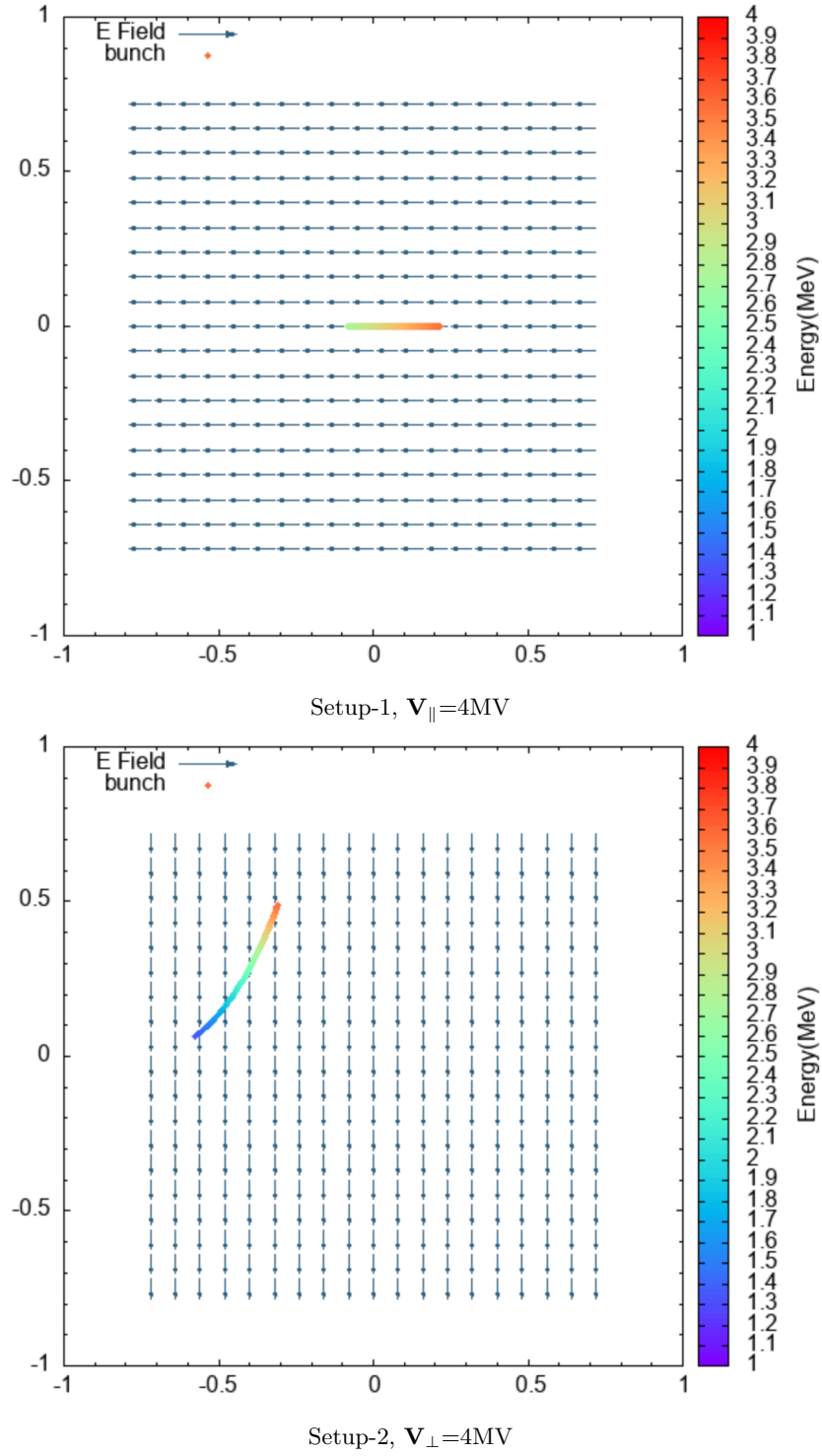


Figure 7: Render of the test setups.  
 $E_{in} = 1 \text{ MeV}$ ,  $t_{end} = 6\text{ns}$

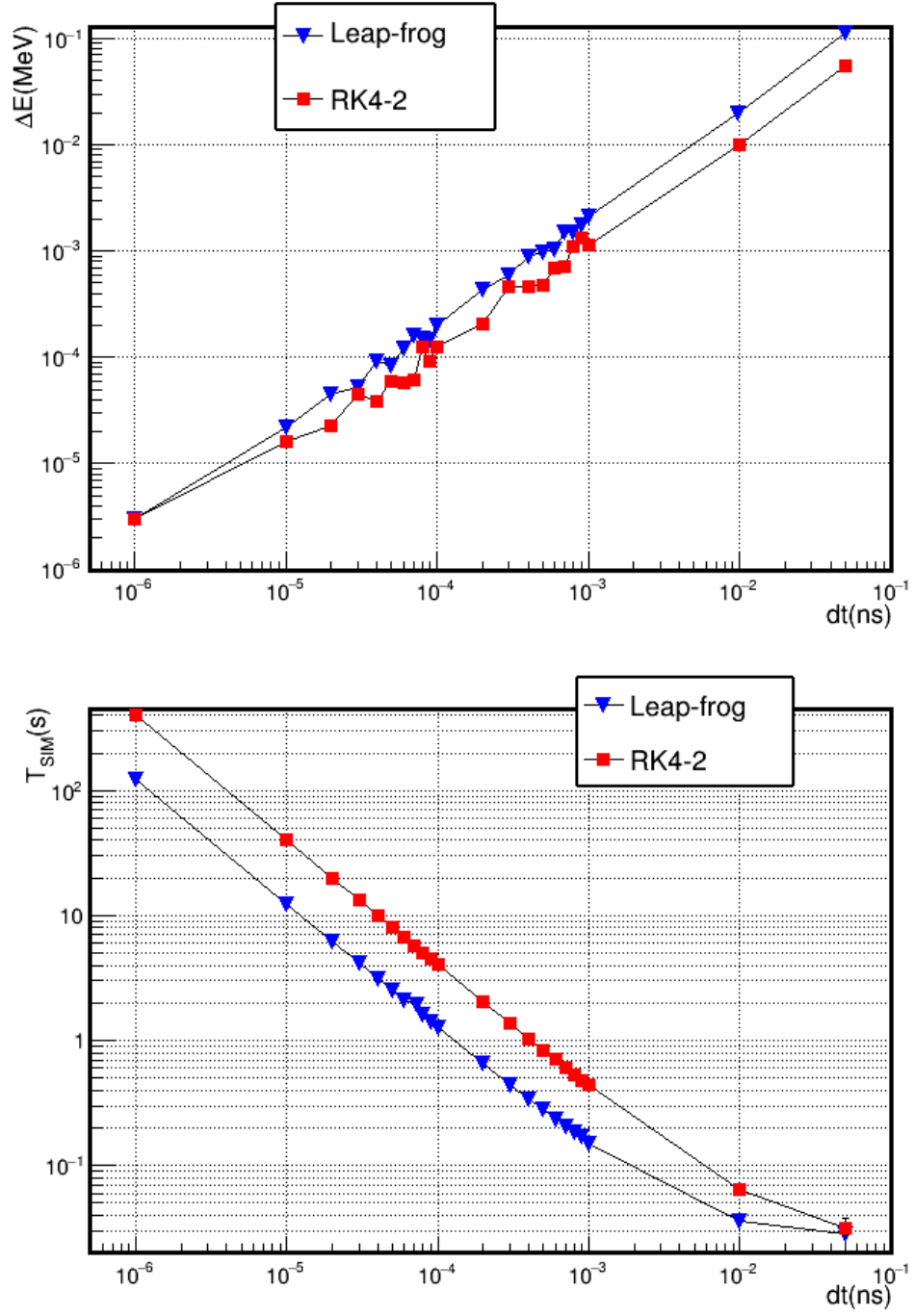


Figure 8: Comparing Leap-frog, RK4-2 performance on  $e^- - \vec{E}$  interaction  
 $E_{in} = 1\text{MeV}$ ,  $V_{\parallel} = 4\text{MV}$ ,  $t_{end} = 6\text{ns}$

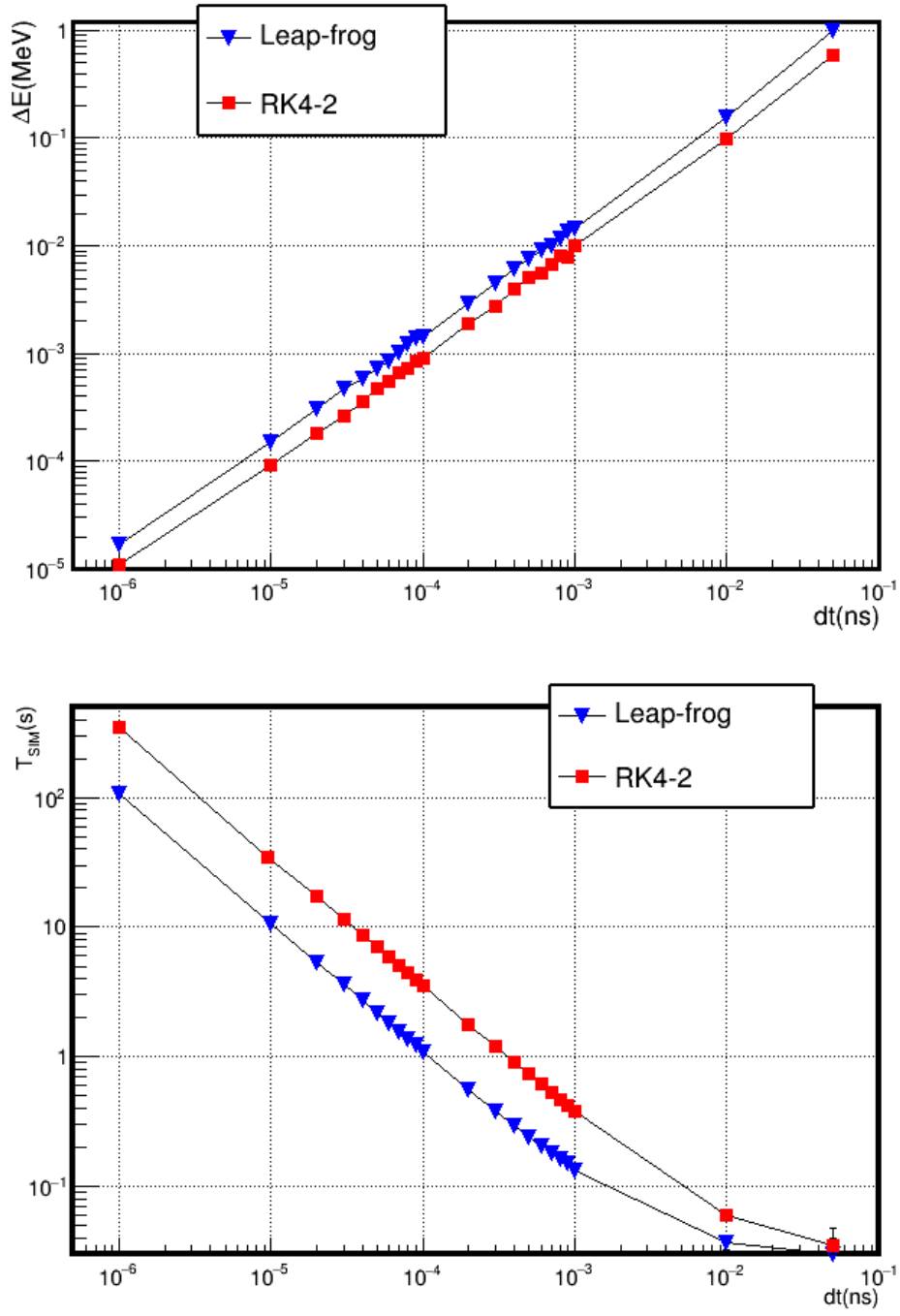


Figure 9: Comparing Leap-frog, RK4-2 performance on  $\mathbf{e}^- - \vec{\mathbf{E}}$  interaction  
 $E_{in} = 1\text{MeV}$ ,  $\mathbf{V}_{\perp} = 4\text{MV}$ ,  $t_{end} = 6\text{ns}$

Taking the  $dt = 10^{-5}$  ns for the reference point as before, the relative performance can be calculated.

$$\begin{aligned}
\Delta E_{LF}^1 &= 22 \times 10^{-6} \text{ MeV} & \Delta E_{LF}^2 &= 15 \times 10^{-5} \text{ MeV} \\
\Delta E_{RK}^1 &= 16 \times 10^{-6} \text{ MeV} & \Delta E_{RK}^2 &= 9.3 \times 10^{-5} \text{ MeV} \\
T_{LF}^1 &= 12.40 \pm 0.04 \text{ s} & T_{LF}^2 &= 10.75 \pm 0.04 \text{ s} \\
T_{RK}^1 &= 40.08 \pm 0.12 \text{ s} & T_{RK}^2 &= 34.93 \pm 0.39 \text{ s}
\end{aligned}$$

$$\begin{aligned}
\Delta E_{LF}^1 \times T_{LF}^1 &= 270 \times 10^{-6} \pm 10^{-6} \text{ MeVs} \\
\Delta E_{RK}^1 \times T_{RK}^1 &= 640 \times 10^{-6} \pm 2 \times 10^{-6} \text{ MeVs} \\
F_{LF}^1 / F_{RK}^1 &= \frac{\Delta E_{RK}^1 \times T_{RK}^1}{\Delta E_{LF}^1 \times T_{LF}^1} = 2.4 \pm 0.02
\end{aligned} \tag{16}$$

$$\begin{aligned}
\Delta E_{LF}^2 \times T_{LF}^2 &= 160 \times 10^{-5} \pm 10^{-5} \text{ MeVs} \\
\Delta E_{RK}^2 \times T_{RK}^2 &= 320 \times 10^{-5} \pm 10^{-5} \text{ MeVs} \\
F_{LF}^2 / F_{RK}^2 &= \frac{\Delta E_{RK}^2 \times T_{RK}^2}{\Delta E_{LF}^2 \times T_{LF}^2} = 2.0 \pm 0.03
\end{aligned} \tag{17}$$

*Leap-frog* provides 2.4 times and 2.0 times less overacceleration per simulation time than *RK4* in parallel and perpendicular electric fields respectably. This results can be tested further with observing from the data (see TODO),

$$\begin{aligned}
\Delta E_{LF}^1(dt = 2 \times 10^{-5}) &= \Delta E_{RK}^1(dt = 3 \times 10^{-5}) &= 45 \times 10^{-6} \text{ MeV} \\
T_{LF}^1(dt = 2 \times 10^{-5}) &= 6.23 \pm 0.02 \text{ s} \\
T_{RK}^1(dt = 3 \times 10^{-5}) &= 13.40 \pm 0.03 \text{ s} \\
F_{LF}^1 / F_{RK}^1 &= \frac{T_{RK}^1(dt = 3 \times 10^{-5})}{T_{LF}^1(dt = 2 \times 10^{-5})} = 2.15 \pm 0.02
\end{aligned} \tag{18}$$

$$\begin{aligned}
\Delta E_{LF}^2(dt = 3 \times 10^{-5}) &\approx \Delta E_{RK}^2(dt = 5 \times 10^{-5}) &\approx 470 \times 10^{-6} \text{ MeV} \\
T_{LF}^2(dt = 3 \times 10^{-5}) &= 3.62 \\
T_{RK}^2(dt = 5 \times 10^{-5}) &= 7.00 \\
F_{LF}^2 / F_{RK}^2 &= \frac{T_{RK}^2(dt = 5 \times 10^{-5})}{T_{LF}^2(dt = 3 \times 10^{-5})} = 1.9 \pm 0.1
\end{aligned} \tag{19}$$

The last uncertainty was taken high due to the approximation of  $467 \approx 471$ . After combining equations 16, 17, 18 and 19, the performance of *Leap-frog* relative to *RK4* in  $\mathbf{e}^- - \vec{\mathbf{E}}$  interaction can be calculated as in equation 20.

$$F_{LF} / F_{RK} = 2.11 \pm 0.03 \tag{20}$$

Therefore, it can be concluded that *Leap-frog* outperforms *RK4* with the relative performance of  $2.11 \pm 0.03$  in  $\mathbf{e}^- - \vec{\mathbf{E}}$  interactions with static uniform  $\vec{\mathbf{E}}$  field.

### 0.1.6 Multithreading

As already mentioned before, multithreading implementation efforts were ongoing since right after the start of this project. There have been a number of different approaches for implementation. First implementation was done when the *Rhodotron Simulation* was only capable of 1D simulations. After the sizable refactoring done for 3D capabilities, this implementation was obsolete.



Figure 10: An Illustration of the multithreading architecture.

For the current version, a main thread that spawns and manages several other worker threads would be used as can be seen in *figure 10*. The UI-Console thread would handle incoming and outgoing communication, notifying the user about the status of simulation (see *figure B.5* in *Appendix B* for example console notification, *figure A.12* in *Appendix A* for implementation) or communicating with the *GUI* that will be discussed in the next section.

Focus of this section is the worker threads, also known as  $e^-$  - **EM** interaction threads. There were four competing architecture for these worker threads,

1. Have a thread pool that calculates  $e^-$  - **EM** in a queue
2. Assign a thread to each bunch
3. Assign random electrons to each thread and calculate  $e^-$  - **EM** with global time
4. Assign random electrons to each thread and calculate  $e^-$  - **EM** with local thread time

**Architecture 1** would require constant waiting in worker threads to get mutexes of  $\vec{E}$  field, especially in *RK4*.

**Architecture 2** was performing well in configurations with a large number of bunches, but was not increasing performance in lower bunch count configurations as expected.

**Architecture 3** was inefficient and wasteful since all the worker threads would wait the main thread to get the next time after they finish calculation, while the main thread would be waiting for the slowest worker thread.

**Architecture 4** was thought to be the best performer. It would give freedom to calculate the whole simulation to each thread while giving up the global time. This would also mean thread-safety is ensured since there is no shared data between the worker threads. However, this architecture can cause issues if  $e^- - e^-$  interactions were decided to be implemented in the future.

Another caveat is that a copy operation of  $\vec{E}$  and  $\vec{B}$  objects for each worker thread would be needed. This would lead to larger memory allocations and more time spent setting up simulations. Therefore, *Architecture 4* is not ideal for fast calculations and when the memory is an important constraint. An implementation that can use both **Architecture 2** & **Architecture 4** when necessary would be a better approach considering these methods; nevertheless, **Architecture 4** was chosen to be implemented.

The implementation can be found in *figures A.8, A.9, A.10 and A.11* in *Appendix A*.

### 0.1.7 Graphical User Interface

Until this point, *Rhodotron Simulation* could be used with a configuration file, defining the problem that would be simulated. An example of this configuration file can be found in *figure B.4* of *Appendix B*. This approach was simple and fast; however, it was not suitable for the average target user since required basic knowledge of command line interface and was not up to modern standards. For this reason, a GUI was decided to be built, using *ROOT framework*. This would also enable *Rhodotron simulation* to make use of analysis tools offered by *ROOT*.

The initial design of the *Rhodotron Simulation GUI* can be observed in *figure 11*.



Load Configuration
Save Configuration
Run Simulation
Render From Log
Quit

### B Field Configuration

1) B=0.0743T R=0.15m r=1.114m

2) B=0.0784T R=0.15m r=1.137m

3) B=0.0861T R=0.15m r=1.184m

4) B=0.0933T R=0.15m r=1.241m

5) B=0.1069T R=0.15m r=1.308m

B(T) 
R(m) 
r(m)

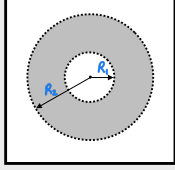
Add
Save
Delete

Rotation of each magnet(degree)

### E Field Configuration

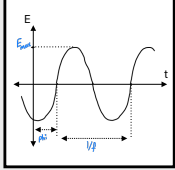
Cavity Description

R1(m)   
R2(m)



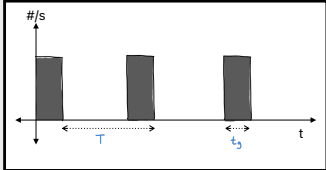
RF Description

Emax(MV/m2)   
f(MHz)   
Phase Lag(degree)



### Gun Configuration

Ein(MeV)



Gun Period - T(ns)

Gun Active Time - tg(ns)

# of bunch

# of e per bunch

### Simulation Configuration

Start time (ns)

End time (ns)

Time step size (ns)

# of threads

☒ Enable multithreading

E field log path

B field log path

Particles log path

Configuration log path

Figure 11: An Illustration of the first GUI design.

The *GUI* would be a standalone application, running the *Rhodotron Simulation* as a service when needed. For this reason, the now called *simulation engine* was updated to be able to run as a background service of *GUI*. By this approach, one could also ignore the *GUI* altogether and use the *simulation engine* as before, as these are two separate products.

In the *figure 12*, the implemented version of *figure 11* can be observed. This version of the *GUI*

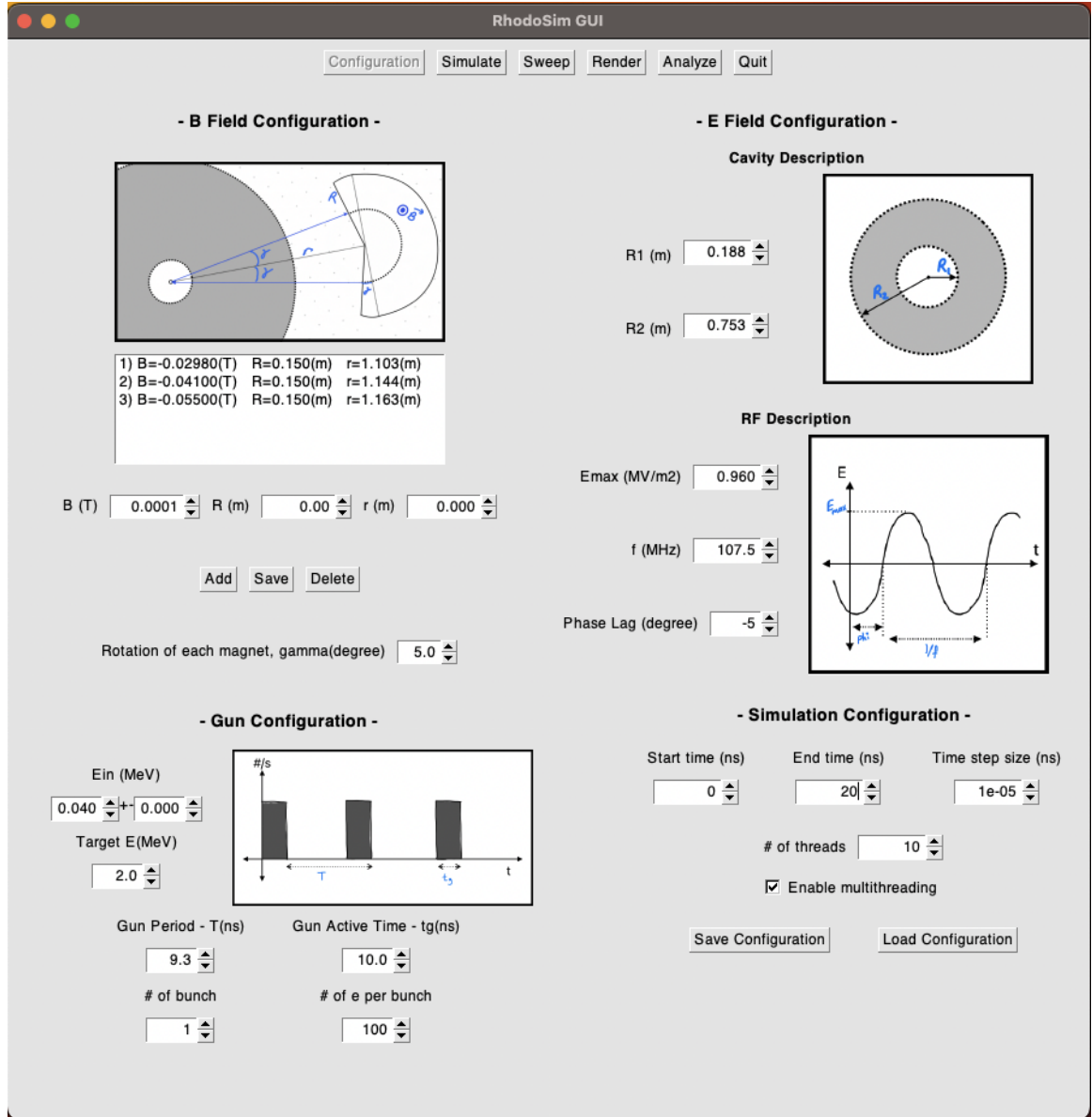
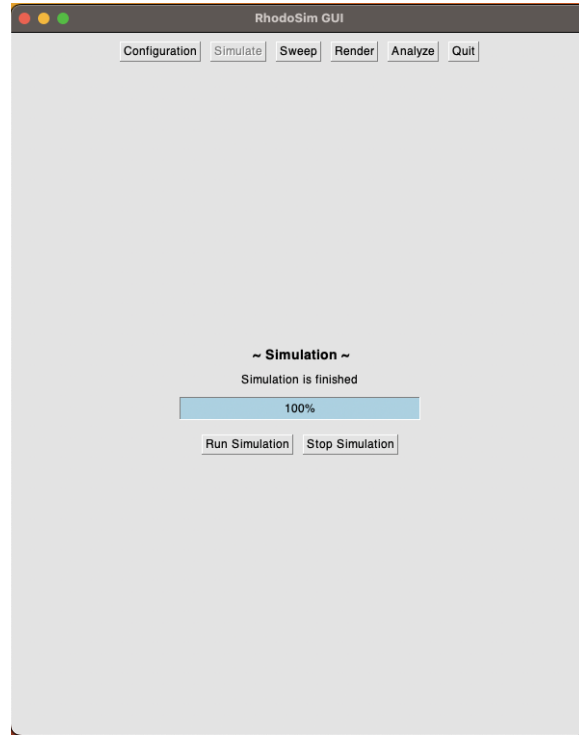


Figure 12: The configuration frame of implemented first GUI design.

Figure 13: Simulate frame of *GUI*.

Configuration Frame provides an interface for specifying, saving or loading a configuration.

Simulation Frame provides an interface for starting and managing the *simulation engine*, displaying the progress of the simulation.

Render Frame provides an interface for rendering the results of a simulation.

Analyze Frame provides an interface for analyzing the results of a simulation.

Sweep Frame provides an interface for parameter sweep functionality.

### Configuration Frame

This frame consists of  $\vec{B}$  field,  $\vec{E}$  field,  $RF$  description,  $e^-$ -gun, *simulation* configuration regions. Each one having an illustration of what the parameters are as can be seen in *figure 12*.

### Simulation Frame

This frame spawns the *simulation engine*, configures and starts the simulation. It has a progress bar that shows the current progress of the simulation, communicating with *UI handler thread* in *simulation engine*.

**Render Frame**

Since the rendering capabilities of *ROOT* is superior than *gnuplot*, the user can render a playback of the simulation in real time, see a specific time frame, export as snapshot or animated gif file.

**Analyze Frame**

Analyze frame provides tools for analyzing and visualizing the simulation data. In the current version **E** distribution histogram and  $\langle E \rangle(t)$  graph of each electron are implemented into this frame. This frame is a work in progress and will be the focus of improvement and become a really powerfull tool in the near future.

**Sweep Frame**

Figure 14: Render frame of *GUI*.



# Appendix A

## Intermediate Codes

---

```
1  for(double i = 2; i < 9; i += dT_out){
2      t_dum += i;
3      double enow = gecis(r_pos, vel, Et, t_dum);
4      if( enow > maxE ){
5          maxE = enow;
6          t_opt = i;
7      }
8      t_dum = t;
9  }
```

---

```
1  double gecis(double r_pos, double vel, double Et, double &t){
2      for(; r_pos >= -R2 && r_pos <= R2 ; t+=dT){
3          vel = c*sqrt(Et*Et-E0*E0)/Et;
4          double RelBeta = vel/c;
5          double RelGamma = 1.0 / sqrt(1.0-RelBeta*RelBeta);
6
7          double ef=Eradial(r_pos*1000,t,RFphase*deg_to_rad);
8
9          double acc=ef*1E6*eQMratio/(RelGamma*RelGamma*RelGamma);
10
11          r_pos = r_pos + vel * dT*ns + 1/2*acc*(dT*ns)*(dT*ns);
12          vel=vel+acc*dT*ns;
13          RelBeta = vel/c;
14          RelGamma = 1.0 / sqrt(1.0-RelBeta*RelBeta);
15          Et=RelGamma*E0;
16      }
17      return Et;
18  }
```

---

Figure A.1:  $L_{out}$  Optimization For Single  $e^-$

---

```

1 int phase_opt(const vector<double>& Louts, int phase_sweep_range){
2     double minrms = 1;
3     int opt_phase;
4     for(int RFphase = -phase_sweep_range; RFphase <= phase_sweep_range; RFphase++){
5         Bunch bunch1(RFphase);
6         double t1 = 0;
7         bunch1.bunch_gecis_t(t1);
8         bunch1.reset_pos();
9
10        for(int i = 0; i < Louts.size(); i++){
11            bunch1.bunch_gecis_d(Louts[i]);
12            bunch1.reset_pos();
13        }
14
15        if( bunch1.E_rms() < minrms ){
16            minrms = bunch1.E_rms();
17            opt_phase = RFphase;
18        }
19    }
20    return opt_phase;
21 }

```

---

Figure A.2:  $\phi_{lag}$  Optimization For Initial Bunch Design

---

```

1 double vector3d::operator* (const vector3d& other){
2     double dot = 0;
3     dot += this->x * other.x;
4     dot += this->y * other.y;
5     dot += this->z * other.z;
6     return dot;
7 }

```

---

```

1 vector3d vector3d::operator% (const vector3d& other){
2     double x_ = (this->y * other.z) - (this->z * other.y);
3     double y_ = (this->z * other.x) - (this->x * other.z);
4     double z_ = (this->x * other.y) - (this->y * other.x);
5     vector3d crossed(x_, y_, z_);
6     return crossed;
7 }

```

---

Figure A.3: \* and % operators of *vector3d* class

---

```

1 bool isInsideHalfSphere(vector3d e_position, double r, vector3d hs_position){
2     vector3d relative = e_position - hs_position;
3     // r/5 can be changed, use this for now
4     if ( relative.magnitude() <= r && relative * hs_position.direction() >= -r/5){
5         return true;
6     }
7     return false;
8 }

```

---

Figure A.4: Logic of is  $e^-$  inside the shape of magnet



---

```

1 vector3d Electron2D::interactB_RK(const MagneticField& B, double time_interval){
2     if (B.isInside(pos) == -1){
3         return vector3d(0,0,0);
4     }
5     Electron2D e_dummy;
6     e_dummy.Et = Et;
7     e_dummy.pos = pos;
8     e_dummy.vel = vel;
9     double time_halved = time_interval*0.5;
10    // get k1
11    vector3d F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
12    vector3d k1 = F_m * e_dummy.gamma_inv();
13    // get k2
14    e_dummy.move(time_halved);
15    e_dummy.accelerate(k1, time_halved);
16    F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
17    vector3d k2 = F_m * e_dummy.gamma_inv();
18    // get k3
19    e_dummy.vel = vel;
20    e_dummy.accelerate(k2, time_halved);
21    vector3d k3 = F_m * e_dummy.gamma_inv();
22    // get k4
23    e_dummy.vel = vel;
24    e_dummy.move(time_halved);
25    e_dummy.accelerate(k3, time_interval);
26    F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
27    vector3d k4 = F_m * e_dummy.gamma_inv();
28
29    return (k1 + k2*2 + k3*2 + k4)/6;
30 }

```

---

Figure A.5: RK4-1 implementation of  $e^- - \vec{B}$ 


---

```

1 void Electron2D::interactRK_ActorE(const RFField& E, const MagneticField& B, double time_interval){
2     vector3d run_kut_E = interactE_RK(E, time_interval);
3     vector3d run_kut_B = interactB_RK(B, time_interval);
4
5     vector3d acc = run_kut_E + run_kut_B;
6
7     move(acc, time_interval/2);
8     accelerate(acc, time_interval);
9     move(acc, time_interval/2);
10 }

```

---

Figure A.6: RK4-1 implementation of  $e^- - \vec{EM}$

---

```

1  void Electron2D::interactRK(RFField& E, MagneticField& B, const double time, double time_interval){
2      Electron2D e_dummy;
3      e_dummy.Et = Et;
4      e_dummy.pos = pos;
5      e_dummy.vel = vel;
6
7      // Calculate k1
8      vector3d acc_E = E.actOn(e_dummy);
9      vector3d acc_B = B.actOn(e_dummy);
10     vector3d acc = acc_E + acc_B;
11     e_dummy.move(time_interval);
12     e_dummy.accelerate(acc, time_interval);
13     vector3d pos_k1 = e_dummy.pos, vel_k1 = e_dummy.vel;
14
15     // Calculate k2
16     e_dummy.pos = (pos + pos_k1)*0.5;
17     e_dummy.vel = (vel + vel_k1)*0.5;
18     e_dummy.Et = e_dummy.gamma()*E0;
19     E.update(time + time_interval*0.5);
20
21     acc_E = E.actOn(e_dummy);
22     acc_B = B.actOn(e_dummy);
23     acc = acc_E + acc_B;
24     e_dummy.move(time_interval);
25     e_dummy.accelerate(acc, time_interval);
26     vector3d pos_k2 = e_dummy.pos, vel_k2 = e_dummy.vel;
27
28     // Calculate k3
29     e_dummy.pos = (pos + pos_k2)*0.5;
30     e_dummy.vel = (vel + vel_k2)*0.5;
31     e_dummy.Et = e_dummy.gamma()*E0;
32     E.update(time + time_interval*0.5);
33
34     acc_E = E.actOn(e_dummy);
35     acc_B = B.actOn(e_dummy);
36     acc = acc_E + acc_B;
37     e_dummy.move(time_interval);
38     e_dummy.accelerate(acc, time_interval);
39     vector3d pos_k3 = e_dummy.pos, vel_k3 = e_dummy.vel;
40
41     // Calculate k4
42     E.update(time + time_interval);
43
44     acc_E = E.actOn(e_dummy);
45     acc_B = B.actOn(e_dummy);
46     acc = acc_E + acc_B;
47     e_dummy.move(time_interval);
48     e_dummy.accelerate(acc, time_interval);
49     vector3d pos_k4 = e_dummy.pos, vel_k4 = e_dummy.vel;
50
51     E.update(time);
52     pos = (pos_k1 + pos_k2*2 + pos_k3*2 + pos_k4)/6;
53     vel = (vel_k1 + vel_k2*2 + vel_k3*2 + vel_k4)/6;
54     Et = gamma()*E0;
55 }

```

---

Figure A.7: RK4-2 implementation of  $e^-$  - **EM**

---

```

1 void RhodotronSimulator::runMT(){
2     gun.fireAllWithFireTimesMT();
3
4     MTEngine.setupPool(time_interval, start_time, end_time, gun, E_field, B_field, gun.thread_bunchs);
5
6     STEPS_TAKEN = 0;
7     simulation_time = start_time;
8     while (simulation_time < end_time + time_interval ){
9         if (STEPS_TAKEN % log_interval() == 0){
10             E_field.update(simulation_time);
11             logEffield(simulation_time, simulation_time + time_interval > end_time);
12             notifyUI(MTEngine.getAverageTime());
13         }
14         simulation_time+=time_interval;
15         STEPS_TAKEN++;
16     }
17     bool end = false;
18     while (!end){
19         double time = MTEngine.getAverageTime();
20         notifyUI(time);
21         if ( time >= end_time ){
22             end = true;
23         }
24         this_thread::yield();
25     }
26     MTEngine.join();
27 }
28

```

---

Figure A.8: Multithreading main-thread logic.

---

```

1 void Gun::fireAllWithFireTimesMT(){
2     std::random_device rd;
3     std::mt19937 e2(rd());
4     std::normal_distribution<double> Edist(Ein, sEin);
5
6     for(_fired_bunch= 0; _fired_bunch < bunch_count; _fired_bunch++){
7         for(_fired_e_in_current_bunch= 0; _fired_e_in_current_bunch < e_per_bunch; _fired_e_in_current_bunch++){
8
9             double E = (sEin == 0) ? Ein : Edist(e2);
10
11             double fire_time = (ns_between_each_electron_fire * _fired_e_in_current_bunch) + _fired_bunch*gun_period;
12
13             auto burrowed_e = bunchs[_fired_bunch].AddElectronGiveAddress(E, gunpos, gundir, fire_time);
14
15             int thread_index = (_fired_e_in_current_bunch + _fired_bunch*e_per_bunch)%thread_bunchs.size();
16
17             thread_bunchs[thread_index]->push_back(burrowed_e);
18         }
19     }
20 }

```

---

Figure A.9: Multithreading electron assign logic.

---

```

1 void MultiThreadEngine::setupPool( double _time_interval, double _start_time, double _end_time, Gun& gun,
2     CoaxialRFfield& RF, MagneticField& B, vector<shared_ptr<vector<shared_ptr<Electron2D>>>>& e_list){
3     threads.reserve(thread_count);
4
5     for(int i = 0; i < thread_count && i == threads.size(); i++){
6
7         child_notifier_mutexes.push_back(make_shared<mutex>());
8         child_times.push_back(make_shared<double>());
9
10         auto _E = RF.Copy();
11         auto _B = B.LightWeightCopy();
12         double time_between_fires = thread_count*gun.getGunActiveTime()/gun.getElectronsPerBunch();
13         double first_fire_time = i*gun.getGunActiveTime()/gun.getElectronsPerBunch();
14         ThreadArguments thread_arguments(i, _time_interval, _start_time, _end_time, &gun, _E, _B, e_list[i], first_fire_time, time_between_fires);
15         thread_arguments.parent_notifier_mutex = child_notifier_mutexes[i];
16         thread_arguments.current_thread_time = child_times[i];
17         threads.push_back(thread(threadLoop, thread_arguments));
18     }
19 }

```

---

Figure A.10: Multithreading worker-threads setup logic.

---

```

1 void threadLoop(ThreadArguments thread_arguments){
2     uint64_t count = 0;
3
4     double sim_time = thread_arguments.start_time;
5
6     while(sim_time < thread_arguments.end_time + thread_arguments.time_interval){
7         thread_arguments.E->update(sim_time);
8         thread_arguments.i_args.time = sim_time;
9
10        if ( count % (unsigned long)(0.1/thread_arguments.time_interval) == 0){
11            saveElectronInfoForSingleThread(thread_arguments.i_args);
12            // Notify the main thread
13            if(thread_arguments.parent_notifier_mutex->try_lock()){
14                *thread_arguments.current_thread_time = sim_time;
15                thread_arguments.parent_notifier_mutex->unlock();
16            }
17        }
18
19        interactForSingleThread(thread_arguments.i_args);
20        // save electron info here
21        sim_time+= thread_arguments.time_interval;
22        count++;
23    }
24    thread_arguments.parent_notifier_mutex->lock();
25    *thread_arguments.current_thread_time = sim_time;
26    thread_arguments.parent_notifier_mutex->unlock();
27 }

```

---

Figure A.11: Multithreading worker-thread logic.

---

```

1  void UIThreadWork(UIThreadArgs args){
2      int UI_WORK_PIECE = SIM_WORK_MASK;
3      if ( !args.isService ) {
4          UI_WORK_PIECE = 50;
5      }
6
7      double piece = (args.end_time - args.start_time)/UI_WORK_PIECE;
8
9      if ( !args.isService ) {
10         std::string sim_running_msg = "...Simulation is running...";
11         for(int i = 0; i < 26 - sim_running_msg.size()/2 ; i++){
12             std::cout << " ";
13         }
14         std::cout << sim_running_msg << "\n";
15     }
16
17     args.ui_mutex->lock();
18     double simtime = *(args.simulation_time);
19     args.ui_mutex->unlock();
20
21     if ( !args.isService ) {
22         std::cout << "V";
23         for(int i = 0; i < 51; i++){
24             std::cout << " _";
25         }
26         std::cout << "V\n[" << std::flush;
27     }
28
29     int count = 0;
30     bool running = true;
31     while(running && (simtime < args.end_time || count < UI_WORK_PIECE )){
32         if( simtime > count * piece ){
33             if ( !args.isService ) {
34                 std::cout << "#" << std::flush;
35             }
36             count++;
37             args.state_mutex->lock();
38             running = *args.state_ptr & SIM_RUNNING;
39             *args.state_ptr = (*args.state_ptr & ~SIM_WORK_MASK) | (count & SIM_WORK_MASK);
40             if (args.isService) write(args._fd, args.state_ptr, SIGNAL_SIZE);
41             args.state_mutex->unlock();
42         }
43         std::this_thread::sleep_for(std::chrono::milliseconds(25));
44         args.ui_mutex->lock();
45         simtime = *(args.simulation_time);
46         args.ui_mutex->unlock();
47     }
48
49     if ( !args.isService ) {
50         std::cout << "#]\n\n" << std::flush;
51         std::cout << "    ...Simulation is finished successfully...\n\n" << std::flush;
52     }
53
54     args.state_mutex->lock();
55     *args.state_ptr |= SIM_RENDERING;
56     if ( args.isService ) write(args._fd, args.state_ptr, SIGNAL_SIZE);
57     args.state_mutex->unlock();
58 }

```

---

Figure A.12: UI-Console handler thread logic.



# Appendix B

## Example Simulation Runs

---

```
1 Optimal phase with the least RMS : -5
2
3 Simulation settings :
4 ph = -5 deg, gt = 1 ns, enum = 1000
5 dT = 0.001 ns, dT_out = 0.01 ns
6
7 For the 1th magnet:
8 Optimum out path = 0.81044 m
9 Magnet guide = 0.25852 m
10 Rho = 0.088477 m
11 Drift time of the first electron in the bunch : 7.688 ns
12 Drift time of the last electron in the bunch : 7.487 ns
13 Max energy = 0.47581 MeV
14 RMS = 0.0058165 MeV
15
16 For the 2th magnet:
17 Optimum out path = 1.0833 m
18 Magnet guide = 0.37766 m
19 Rho = 0.098898 m
20 Drift time of the first electron in the bunch : 5.597 ns
21 Drift time of the last electron in the bunch : 5.617 ns
22 Max energy = 0.89172 MeV
23 RMS = 0.0099018 MeV
24
25 For the 3th magnet:
26 Optimum out path = 1.1705 m
27 Magnet guide = 0.41573 m
28 Rho = 0.10223 m
29 Drift time of the first electron in the bunch : 5.314 ns
30 Drift time of the last electron in the bunch : 5.325 ns
31 Max energy = 1.298 MeV
32 RMS = 0.013879 MeV
33
34 Electron with the most energy : 623) 1.6999 MeV,          RMS of bunch : 0.017981 MeV
35
36 Total steps calculated : 12468052652
37 Simulation finished in : 632050015 us      ( 632.1 s )
38
```

---

Figure B.1:  $\phi_{lag}$ ,  $\rho$  &  $L$  optimization at  
 $P = 12\text{KW}$ ,  $R_1 = 0.188\text{m}$ ,  $R_2 = 0.753\text{m}$ ,  $t_g = 1\text{ns}$ ,  $E_{in} = 40\text{KeV}$

---

```

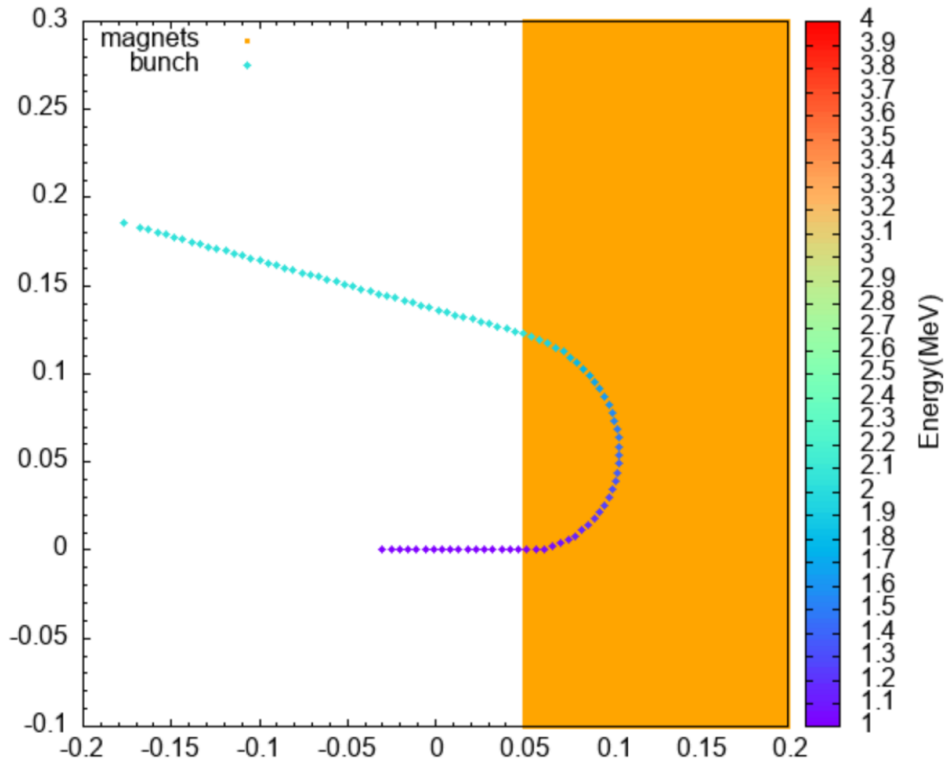
1  Optimal phase with the least RMS : 0
2
3  Simulation settings :
4  ph = 0 deg, gt = 0.8 ns, enum = 1000
5  dT = 0.001 ns, dT_out = 0.01 ns
6
7  For the 1th magnet:
8  Optimum out path = 0.80787 m
9  Magnet guide = 0.2574 m
10 Rho = 0.088379 m
11 Drift time of the first electron in the bunch : 7.629 ns
12 Drift time of the last electron in the bunch : 7.48 ns
13 Max energy = 0.47579 MeV
14 RMS = 0.0038689 MeV
15
16 For the 2th magnet:
17 Optimum out path = 1.0833 m
18 Magnet guide = 0.37765 m
19 Rho = 0.098898 m
20 Drift time of the first electron in the bunch : 5.589 ns
21 Drift time of the last electron in the bunch : 5.605 ns
22 Max energy = 0.89169 MeV
23 RMS = 0.0068848 MeV
24
25 For the 3th magnet:
26 Optimum out path = 1.1705 m
27 Magnet guide = 0.41573 m
28 Rho = 0.10223 m
29 Drift time of the first electron in the bunch : 5.311 ns
30 Drift time of the last electron in the bunch : 5.318 ns
31 Max energy = 1.298 MeV
32 RMS = 0.0096887 MeV
33
34 Electron with the most energy : 629) 1.6999 MeV,          RMS of bunch : 0.012318 MeV
35
36 Total steps calculated : 12455378454
37 Simulation finished in : 631136046 us      ( 631.1 s )

```

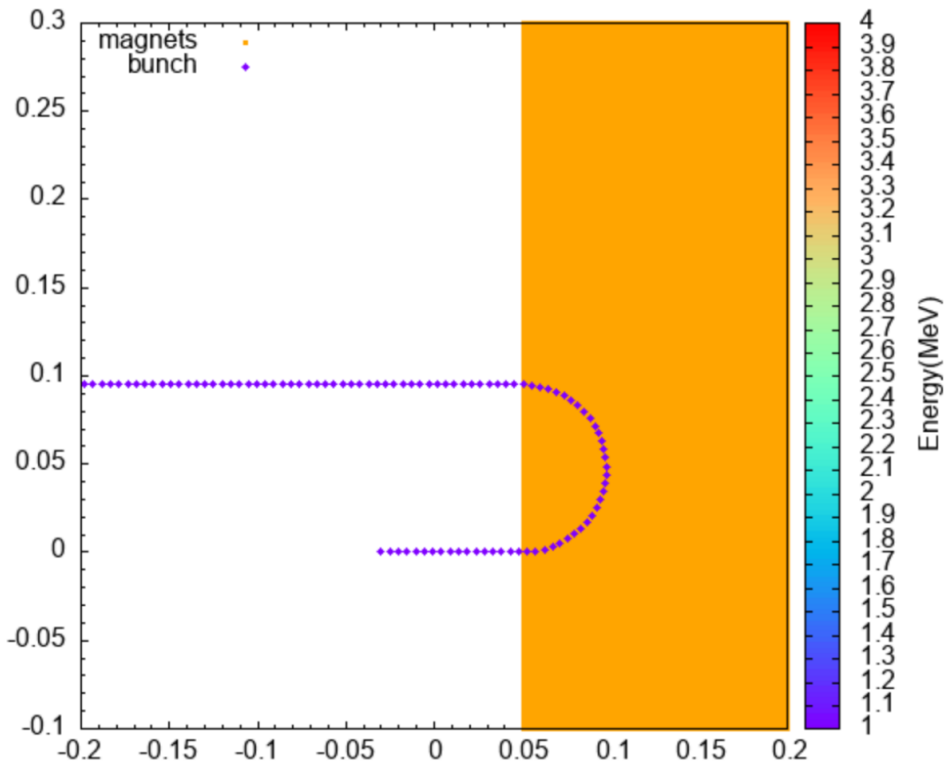
---

Figure B.2:  $\phi_{lag}$  &  $\rho$  &  $L$  optimization at  
 $P = 12\text{KW}$ ,  $R_1 = 0.188\text{m}$ ,  $R_2 = 0.753\text{m}$ ,  $t_g = 0.8\text{ns}$ ,  $E_{in} = 40\text{KeV}$





$dt = 10^{-2}\text{ns}$ ,  $\Delta E = 1.047\text{MeV}$



$dt = 10^{-4}\text{ns}$ ,  $\Delta E = 0.006\text{MeV}$

Figure B.3: Energy gain of 1MeV bunch in  $\mathbf{B}=0.1\text{T}$  using RK4-2

---

```

1  # Rhodotron Simulation Configuration File
2  # =====
3  # M.Furkan Er 22/09/2022
4  # =====
5  #
6  # emax = Maximum electric field strength (MV/m)
7  # ein = Energy of electrons coming out of the gun (MeV)
8  # einstd = Standard deviation of energy of electrons coming out of the gun (MeV)
9  # targeten = Max energy on the output gif (MeV)
10 # freq = Frequency of the RF field (MHz)
11 # phaselag = phase lag of the first electrons (degree)
12 # starttime = time to start firing the gun (ns)
13 # endtime = ns to run the simulation (ns)
14 # dt = time interval to do the calculations (ns)
15 # guntime = how long a gun pulse is (ns)
16 # gunperiod = time between two gun pulses (ns)
17 # enum = number of electrons to simulate in a bunch
18 # bunchnum = number of times the gun fires
19 # r1 = radius of the inner cylinder (m)
20 # r2 = radius of the outer cylinder (m)
21 # epath = path to store the electric field data
22 # bpath = path to store the magnetic field data
23 # cpath = path to store the settings
24 # ppath = path to store electron data
25 # multh = enable or disable multithreading
26 # thcount = set the maximum thread to be used
27 # magrotation = degrees of rotation to enter each magnet
28 # addmagnet = takes 3 input. (B , R, < Radial distance of center >)
29 # output = output file name
30
31
32 # E FIELD CONFIGURATION
33 emax=1.170
34 freq=107.3
35 phaselag=10.0
36 r1=0.1840
37 r2=0.7380
38
39 # B FIELD CONFIGURATION
40 magrotation=5.0
41
42 # GUN CONFIGURATION
43 einmean=0.040
44 einstd=0.0000
45 targeten=2.0
46 guntime=1.0
47 gunperiod=9.3
48 enum=50
49 bunchnum=1
50
51 # SIM CONFIGURATION
52 starttime=0
53 endtime=10
54 dt=0.0000010000
55 epath=xy/ef.dat
56 bpath=xy/magnet.dat
57 cpath=xy/settings.dat
58 ppath=xy/paths/
59 multh=1
60 thcount=10

```

---

Figure B.4: An example *config.in* file.

---

```

1  -- Simulation Configuration --
2  Emax : 0.96    MV/m
3  Freq : 107.5   MHz
4  Phase Lag : -5 degree
5  EndTime : 100  ns
6  dT : 0.0001   ns
7  guntime : 0.8  ns
8  gunperiod : 9.3 ns
9  enum : 100
10 bunchnum : 2
11 R1 : 0.188241  m
12 R2 : 0.752967  m
13 Magnet count : 5
14 Ein : 0.04     MeV
15 TargetE : 2.5  MeV
16 -----
17
18      ...Simulation is running...
19 V-----V
20 [#####]

```

---

Figure B.5: Example of console output while simulation is running.





## Appendix C

## Data and Graphs

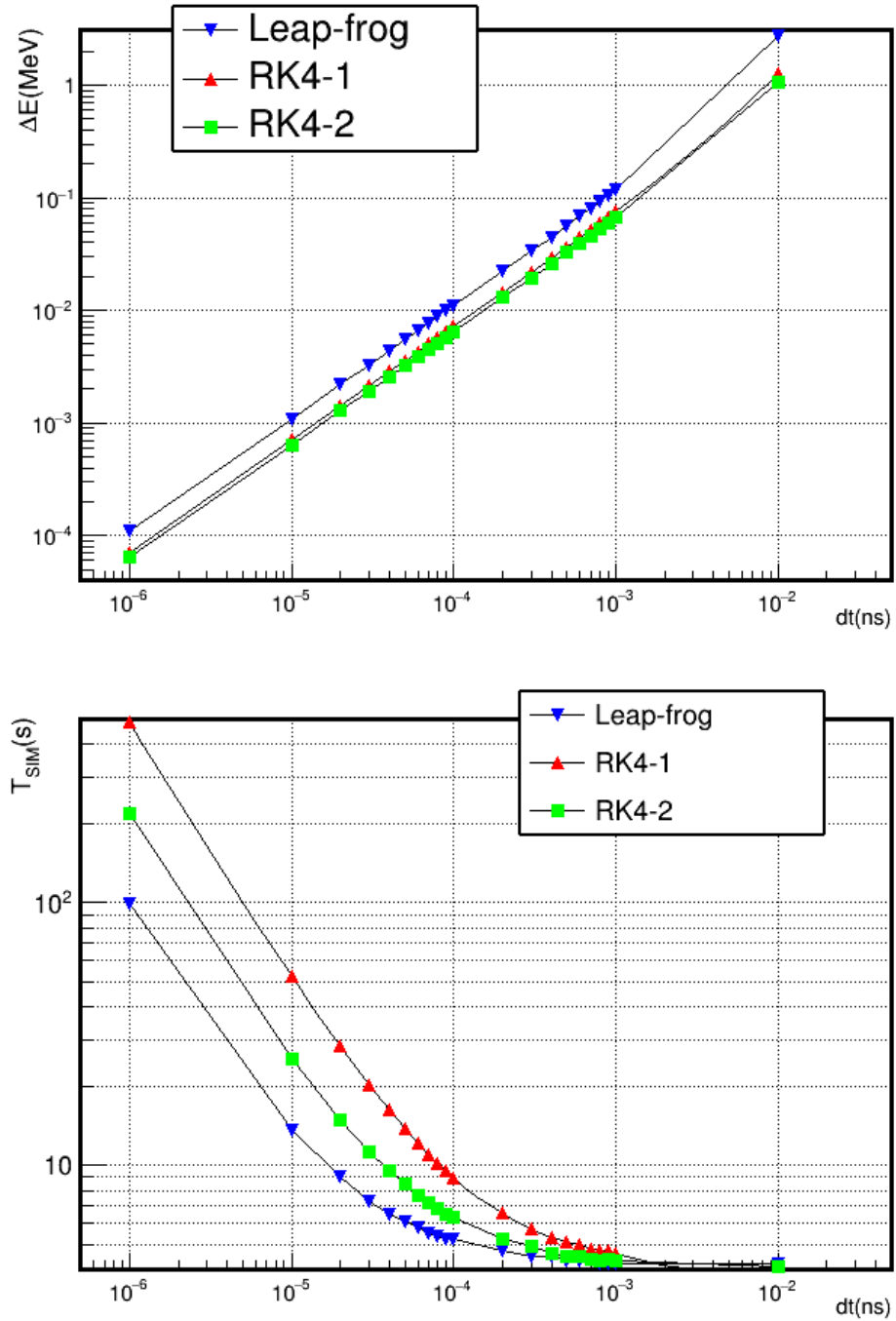


Figure C.1: Comparing Leap-frog, RK4-1, RK4-2 performance on  $\sigma^- - \vec{B}$  interaction

dt(ns)	$\Delta E_{avg}$ (MeV)	$\mu T_{sim}$ (s)	$\sigma T_{sim}$ (s)
1e-02	2.783228	0.034660	0.011537
1e-03	0.117124	0.115775	0.001071
9e-04	0.104552	0.128983	0.001820
8e-04	0.092252	0.143251	0.002585
7e-04	0.080144	0.158808	0.003228
6e-04	0.068258	0.182359	0.002426
5e-04	0.056554	0.215104	0.002807
4e-04	0.044931	0.262952	0.005119
3e-04	0.033467	0.341552	0.002610
2e-04	0.022158	0.501784	0.005709
1e-04	0.011006	0.973145	0.005849
9e-05	0.009899	1.084032	0.010985
8e-05	0.008792	1.216145	0.012486
7e-05	0.007688	1.387908	0.019031
6e-05	0.006586	1.604475	0.011775
5e-05	0.005485	1.926505	0.014535
4e-05	0.004384	2.395898	0.009702
3e-05	0.003286	3.178265	0.014099
2e-05	0.002189	4.740706	0.022709
1e-05	0.001094	9.441138	0.027266
1e-06	0.000109	93.888320	0.290820

Table C.1: *Leap-frog* data on  
 $E_{in} = 1\text{MeV}$ ,  $\mathbf{B}=0.1\text{T}$ ,  $t_{end} = 5\text{ns}$

dt(ns)	$\Delta E_{avg}$ (MeV)	$\mu T_{sim}$ (s)	$\sigma T_{sim}$ (s)
1e-02	1.047130	0.048943	0.011642
1e-03	0.066912	0.239299	0.003483
9e-04	0.059899	0.268007	0.004530
8e-04	0.053028	0.296154	0.004146
7e-04	0.046183	0.333123	0.004259
6e-04	0.039452	0.384046	0.002458
5e-04	0.032734	0.456387	0.003888
4e-04	0.026072	0.563011	0.004803
3e-04	0.019474	0.742440	0.006169
2e-04	0.012926	1.103559	0.007649
1e-04	0.006437	2.178779	0.009733
9e-05	0.005791	2.411302	0.012266
8e-05	0.005145	2.704117	0.012683
7e-05	0.004500	3.078304	0.013281
6e-05	0.003856	3.589154	0.014472
5e-05	0.003212	4.297561	0.009546
4e-05	0.002568	5.369322	0.012127
3e-05	0.001925	7.136687	0.007845
2e-05	0.001283	10.679166	0.012126
1e-05	0.000641	21.325229	0.011661
1e-06	0.000064	212.824121	0.040967

Table C.2:  $RK4-2$  data on  
 $E_{in} = 1\text{MeV}$ ,  $\mathbf{B}=0.1\text{T}$ ,  $t_{end} = 5\text{ns}$