0.1 Intermediate Versions

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

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 $\mathbf{e}^-$ outside of the cavity for $t_{out} = L_{out}/v$, then inject it back to the cavity with reversed $\vec{\mathbf{v}}$. 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 ϕ_{lag} Optimization For Bunches

After successfully accelerating single $\mathbf{e}^-$, particle bunches were implemented to approximate a real $\mathbf{e}^-$ gun. They were modeled as N electrons fired from an $\mathbf{e}^-$ 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 $\mathbf{e}^-$ gun design.

Addition of bunches would immediately proven useful when finding optimal gun phase lag. ϕ_{lag} for a bunch was defined as the RF phase when the first $\mathbf{e}^-$ 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:

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

Optimal ϕ_{lag} would produce maximum μE 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 ϕ_{lag} sweep was based on minimizing E_{rms} during simulation (*see figure 1*). For μE considerations, data from the software would be analyzed either manually or by using external tools such as *ROOT*.

```cpp
int phase_opt(int phase_sweep_range){
    double minrms = 1;
    int opt_phase;
    for(int RFphase = -phase_sweep_range; RFphase <= phase_sweep_range; RFphase++){
      Bunch bunch1(RFphase);
      double t1 = 0;
      bunch1.bunch_gecis_t(t1);
      bunch1.reset_pos();

      if( bunch1.E_rms() < minrms ){
        minrms = bunch1.E_rms();
        opt_phase = RFphase;
      }
    }
    return opt_phase;
}
```

Figure 1: ϕ_{lag} Optimization For Initial Bunch Design

Since ϕ_{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 ϕ_{lag} in case of production issues in magnets.

After the bunch and ϕ_{lag} sweep implementations, L_{out} sweep was also updated to minimize E_{rms}. ρ 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*.

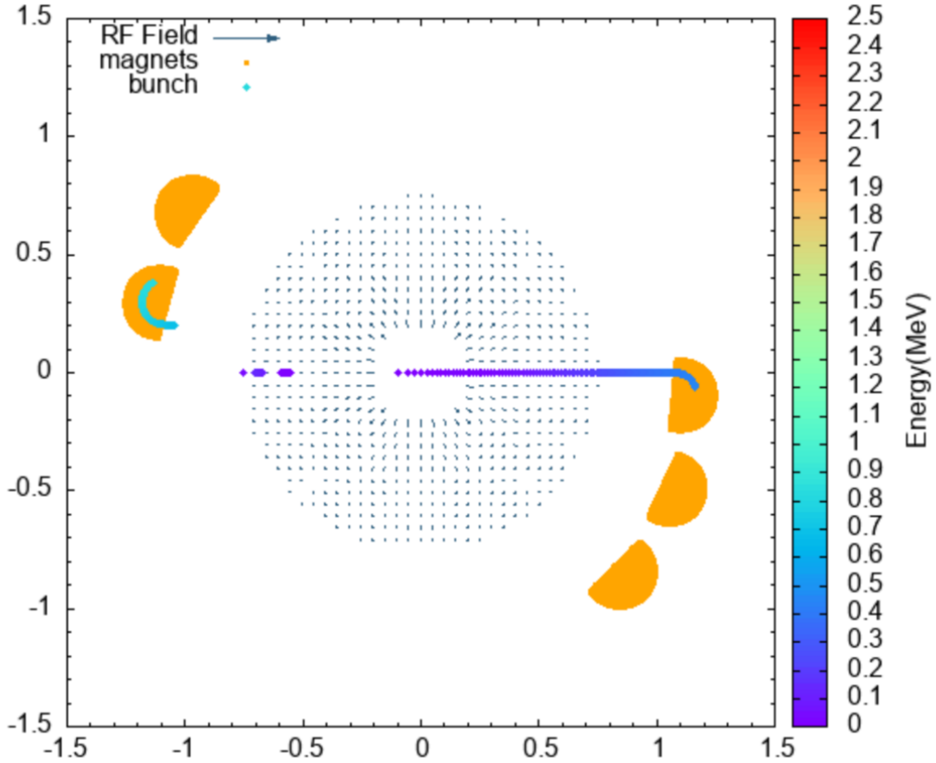
```
vector3d CoaxialRFField::actOn(Electron& e){
    vector3d Efield = getField(e.pos);                          // Calculate E vector
    vector3d F_m = Efield*1E6*eQMratio;                         // Calculate F/m vector
    vector3d acc = (F_m - e.vel*(e.vel*F_m)/(c*c))/e.gamma();   // Calculate a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

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

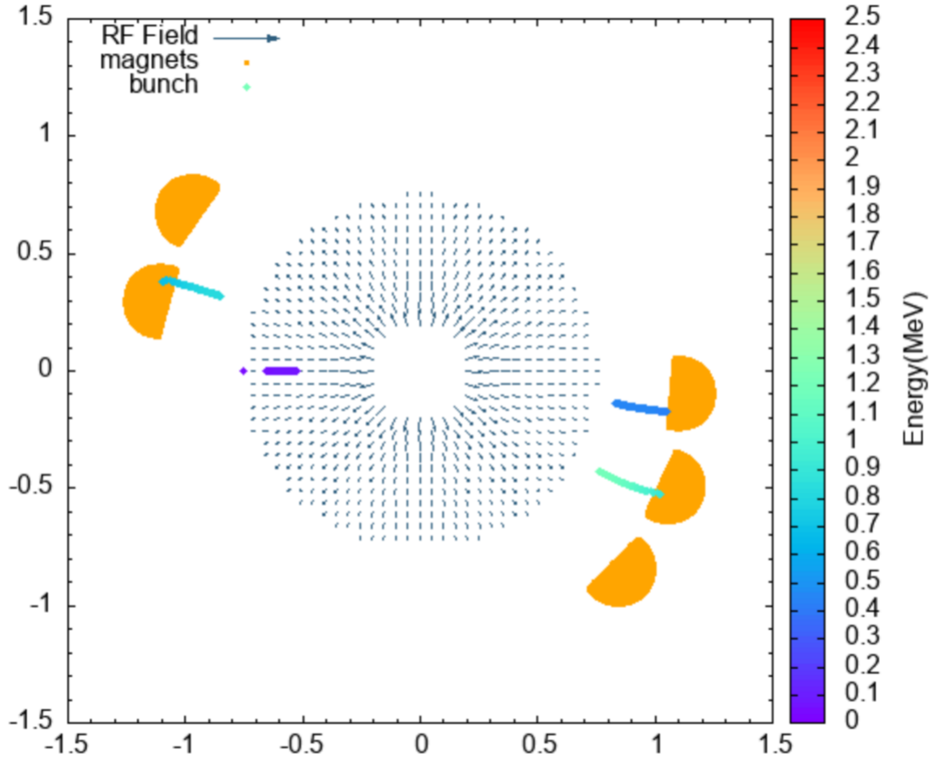
Figure 2: $\mathbf{e}^-$ - **EM** interaction logic from *equation* **??**

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*.



Unsyncronized $\mathbf{e}^-$ gun period, $T_g = 5ns$

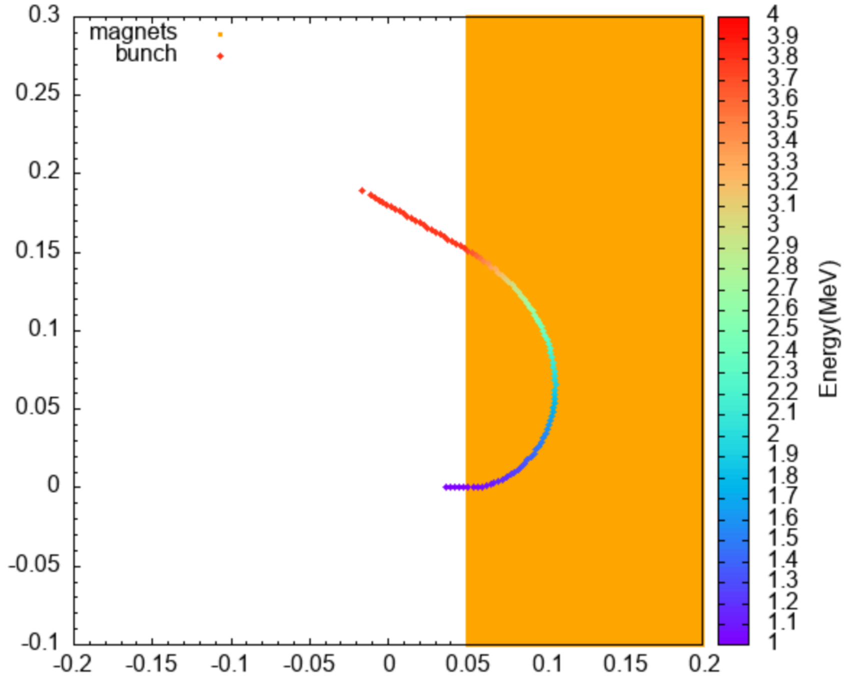


Syncronized $\mathbf{e}^-$ gun period, $T_g = 9.3ns$

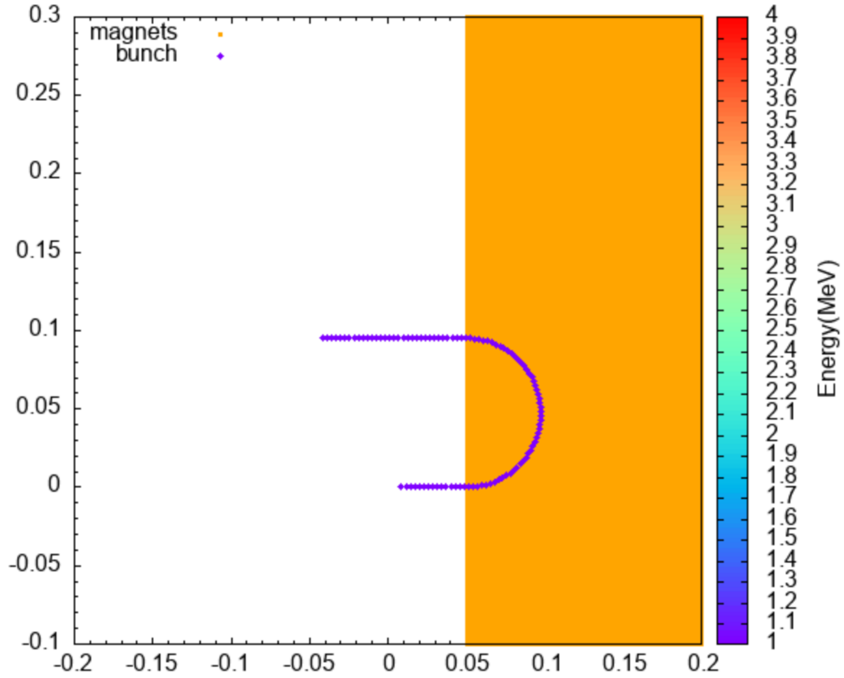
Figure 3: Example *gnuplot* renders of *Rhodotron Simulation*
$P = 12\text{kW},\ f = 107.5\text{MHz}$

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 $x > 0.05$m. Initial results at $dt = 0.01$ns proved the suspicion of $\mathbf{e}^-$ - $\vec{\mathbf{B}}$ interaction being broken. However, the energy gain would decrease tremendously as dt decreased.



$dt = 10^{-2}$ns, $\Delta E = 2.783$MeV



$dt = 10^{-4}$ns, $\Delta E = 0.011$MeV

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

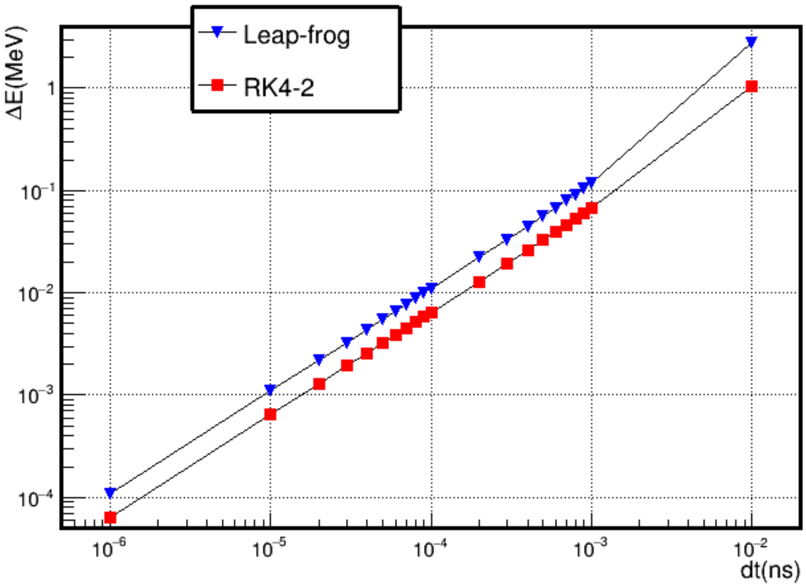
First approach for integrating *section* **??** into $\mathbf{e}^-$ - **EM** interaction was to calculate $\vec{\mathbf{a}}_E$ and $\vec{\mathbf{a}}_B$ from *equation* **??** using *RK4*. After $\vec{\mathbf{a}}_{EM} = \vec{\mathbf{a}}_E + \vec{\mathbf{a}}_B$ was calculated, $\mathbf{e}^-$ would move 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

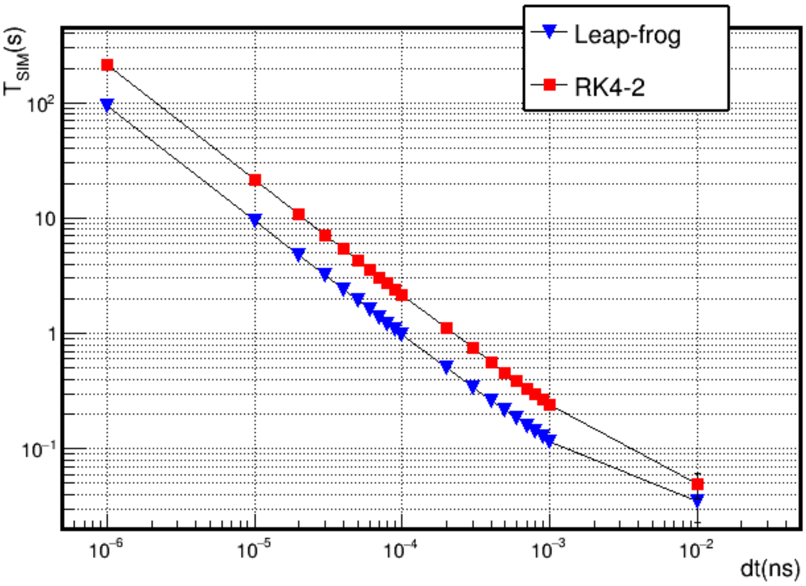
Following the implementation of *RK4-1*, revisions were made to the integration method for *RK4* to replace *Leap-frog.* Instead of calculating $\vec{\mathbf{a}}_{EM}$ using *RK4*, $\vec{\mathbf{r}}$ and $\vec{\mathbf{v}}$ would be determined directly.

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 rigerous 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*.



dt vs ΔE



dt vs T_{sim}

Figure 5: Comparing Leap-frog, RK4-2 performance on $\mathbf{e}^-$ - $\vec{\mathbf{B}}$ interaction
$E_{in} = 1$MeV, $\mathbf{B}$=0.1T, $t_{end} = 5$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 $dt = 10^-5$ns was taken as reference point due to providing a good balance of accuracy and computational intensity,

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

Also, observing from the data,

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

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

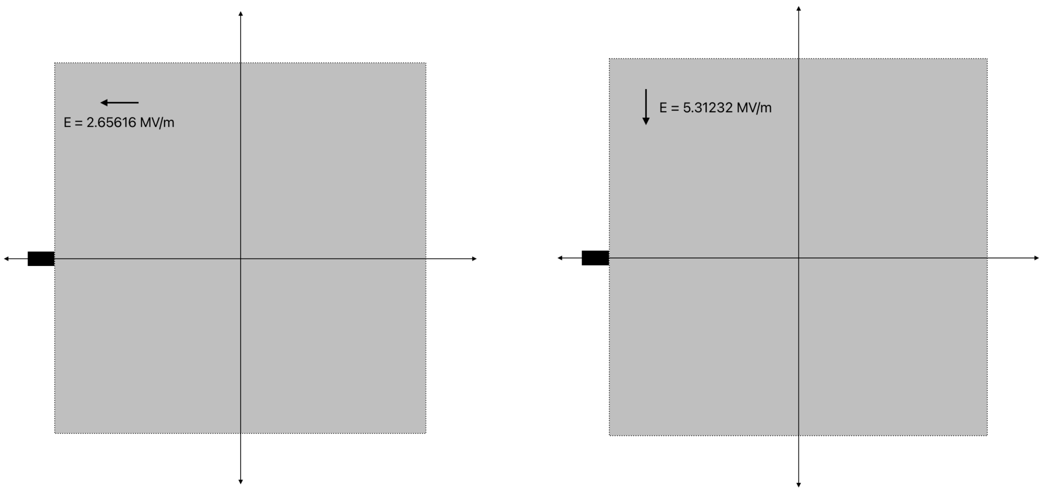
$$F_{LF}/F_{RK} = 1.36 \pm 0.05 \quad (3)$$

Therefore, *Leap-frog* was found to be the better choice as it provided with 1.36 ± 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 confgurations 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 $(-0.753, 0, 0)$m, directed at $(1, 0, 0)$ firing electrons with the kinetic energy of 1MeV.



Setup-1 parallel $\vec{\mathbf{E}}$ field　　Setup-2, perpendicular $\vec{\mathbf{E}}$ field

Figure 6: Illustration of test setups.

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

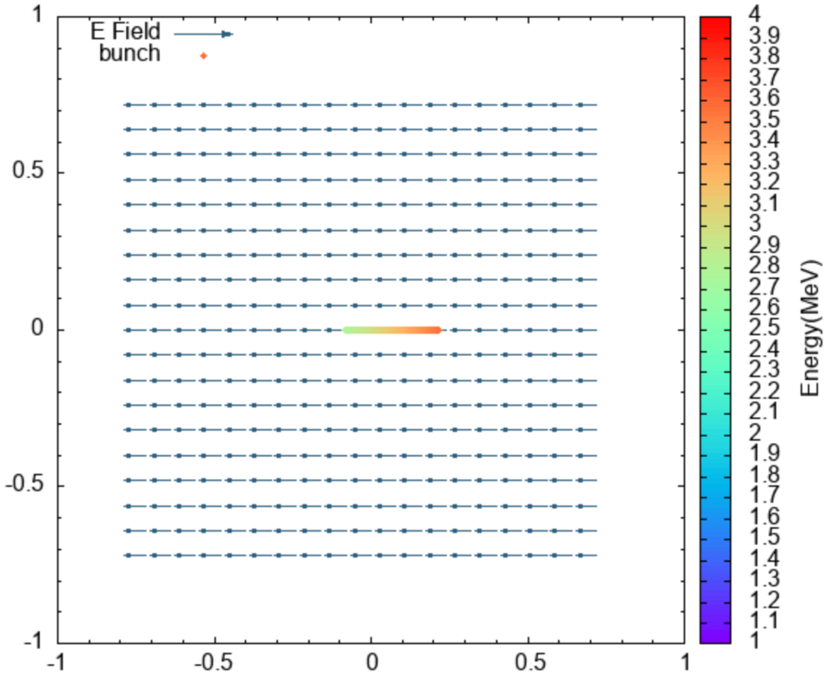
Considering the $\vec{\mathbf{E}}$ is parallel to the beam path, potential difference V in the trejectory until (-0.753, 0, 0)m is

$$\begin{aligned}
\Delta V^1 &= -\int \vec{\mathbf{E}} \cdot \vec{\mathbf{ds}} && (4)\\
&= -\int_{-0.753}^{0.753} -2.65616 \times dx && (5)\\
&= 2.65616 \times 1.506 && (6)\\
\Delta V^1 &= 4MV && (7)\\
\Delta E^1 &= 4MeV && (8)\\
E^1_{exitTH} &= 5MeV && (9)
\end{aligned}$$

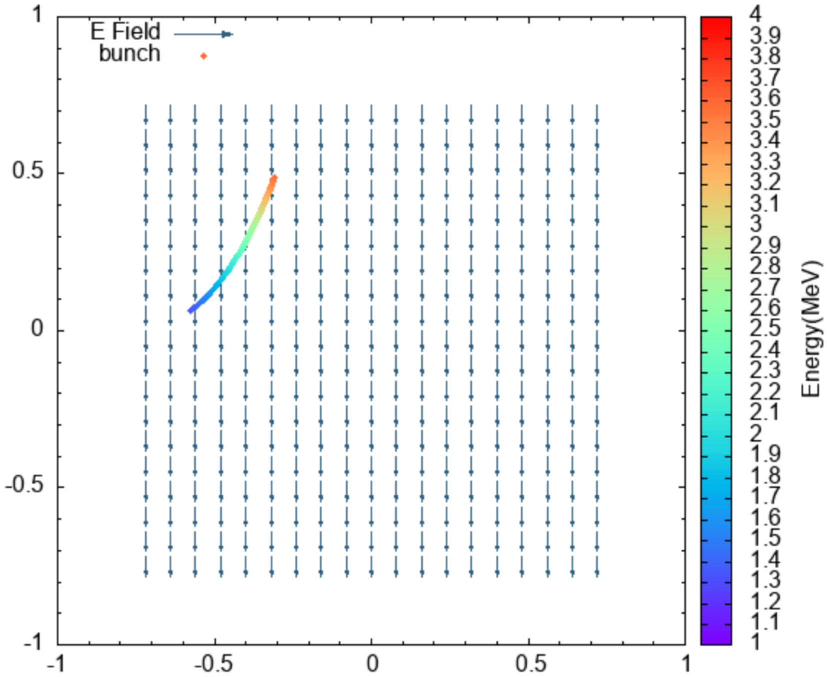
In the second test on the other hand, the beam would be injected into a different static uniform electric field,
$\vec{\mathbf{E}} = (0, -5.31232, 0)$ MV/m where $-0.753 < x < 0.753$ and $-0.753 < y < 0.753$, $\vec{\mathbf{E}} = 0$ elsewhere.

$$\begin{aligned}
\Delta V^2 &= -\int \vec{\mathbf{E}} \cdot \vec{\mathbf{ds}} && (10)\\
&= -\int_{0}^{0.753} -5.31232 \times dy && (11)\\
&= 5.31232 \times 0.753 && (12)\\
\Delta V^2 &= 4MV && (13)\\
\Delta E^2 &= 4MeV && (14)\\
E^2_{exitTH} &= 5MeV && (15)
\end{aligned}$$

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.



Setup-1, $\mathbf{V}_{\parallel}$=4MV



Setup-2, $\mathbf{V}_{\perp}$=4MV

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

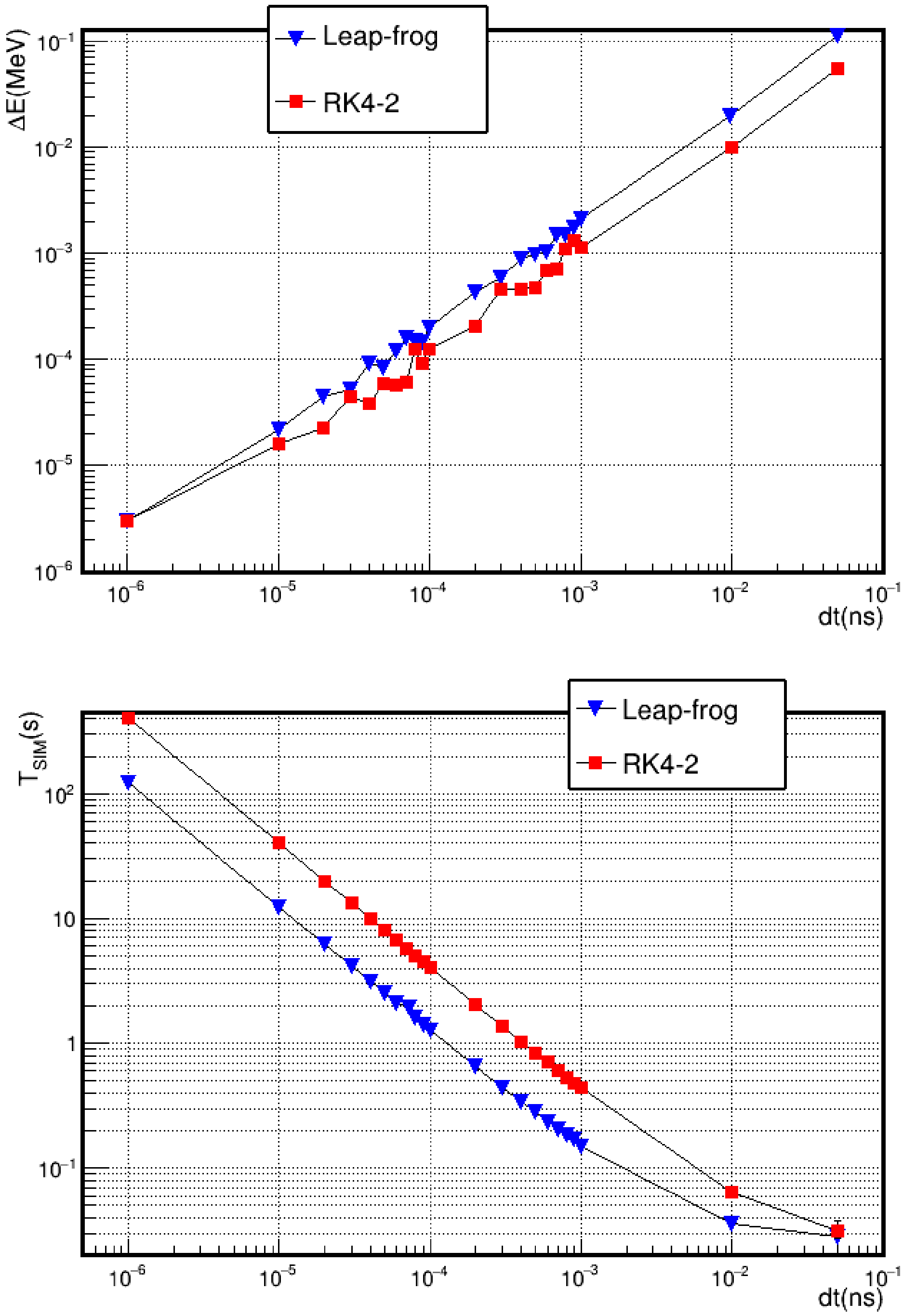


Figure 8: Comparing Leap-frog, RK4-2 performance on $\mathbf{e}^{-}$ - $\vec{\mathbf{E}}$ interaction E_{in} = 1MeV, $\mathbf{V}_{\parallel}$=4MV, t_{end} = 6ns

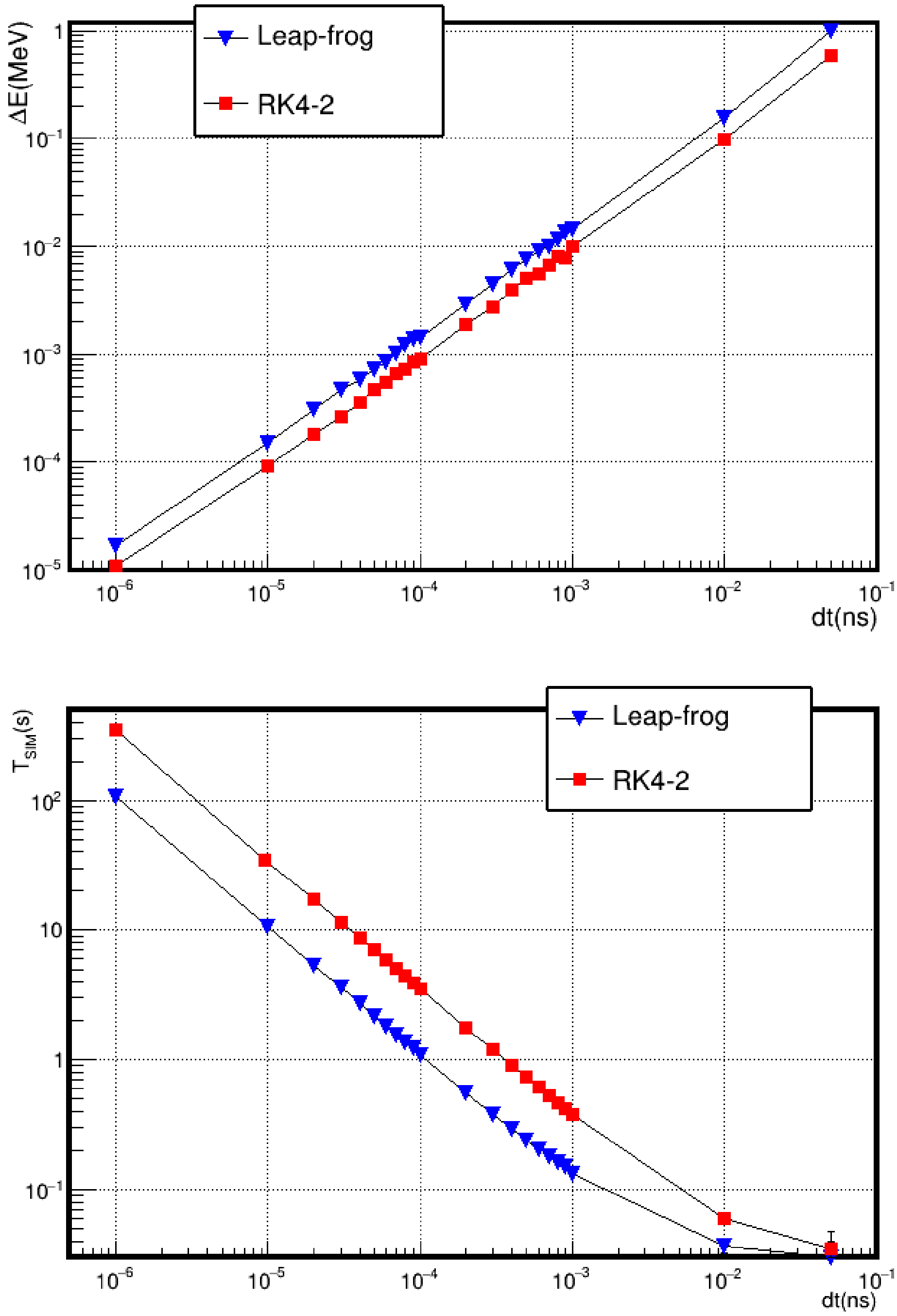


Figure 9: Comparing Leap-frog, RK4-2 performance on $\mathbf{e}^-$ - $\vec{\mathbf{E}}$ interaction
E_{in} = 1MeV, $\mathbf{V}_{\perp}$=4MV, t_{end} = 6ns

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

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

$$\begin{aligned} \Delta E^1_{LF} \times T^1_{LF} &= 270 \times 10^{-6} \pm 10^{-6} MeVs \\ \Delta E^1_{RK} \times T^1_{RK} &= 640 \times 10^{-6} \pm 2 \times 10^{-6} MeVs \\ F^1_{LF}/F^1_{RK} &= \frac{\Delta E^1_{RK} \times T^1_{RK}}{\Delta E^1_{LF} \times T^1_{LF}} = 2.4 \pm 0.02 \end{aligned} \tag{16}$$

$$\begin{aligned} \Delta E^2_{LF} \times T^2_{LF} &= 160 \times 10^{-5} \pm 10^{-5} MeVs \\ \Delta E^2_{RK} \times T^2_{RK} &= 320 \times 10^{-5} 4 \pm 10^{-5} MeVs \\ F^2_{LF}/F^2_{RK} &= \frac{\Delta E^2_{RK} \times T^2_{RK}}{\Delta E^2_{LF} \times T^2_{LF}} = 2.0 \pm 0.03 \end{aligned} \tag{17}$$

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 obsering from the data (see TODO),

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

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

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 ± 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 implemenation was done when the *Rhodotron Simulation* was only capable of 1D simulations. After the sizable refactoring done for 3D capabilities, this implementation was obselete.

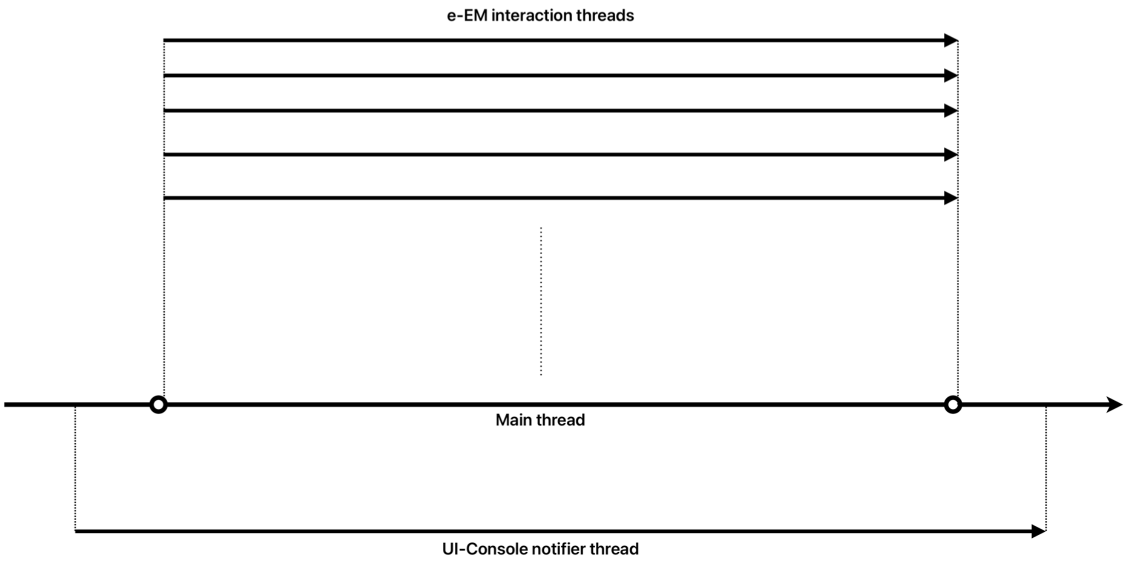


Figure 10: An Illustration of the multithreading architecture.

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 implemenation) or communucating with the *GUI* that will be disscussed in the next section.

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

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

Architecture 1 would require constant waiting in worker threads to get mutexes of $\vec{\mathbf{E}}$ 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 $\mathbf{e}^-$ - $\mathbf{e}^-$ interactions were decided to be implemented in the future.

Another caveat is that a copy operation of $\vec{\mathbf{E}}$ and $\vec{\mathbf{B}}$ 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 chosed to be implemented.

The implemenation 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 configration 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*.

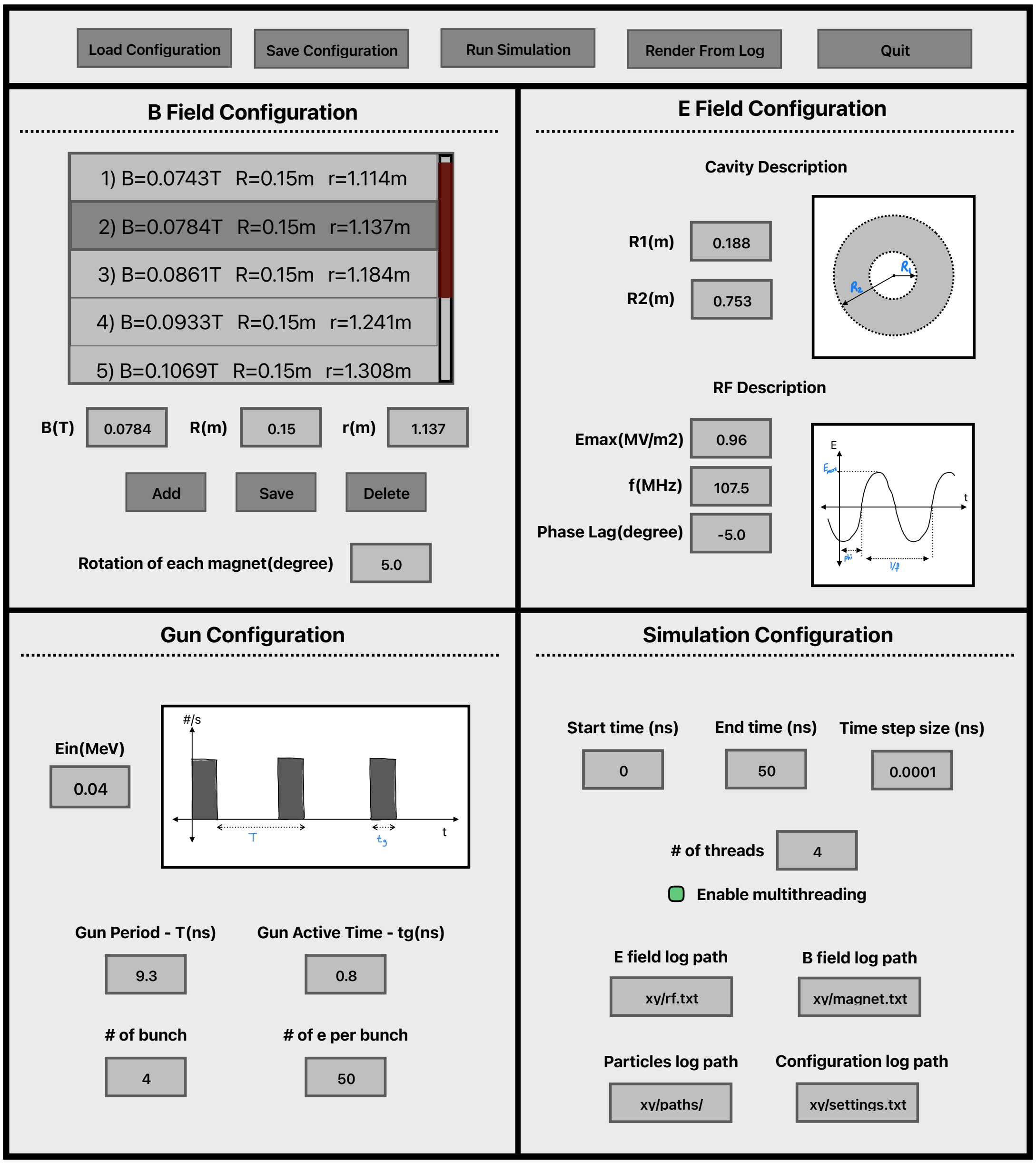


Figure 11: An Illustration of the first GUI design.

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 seperate products.

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

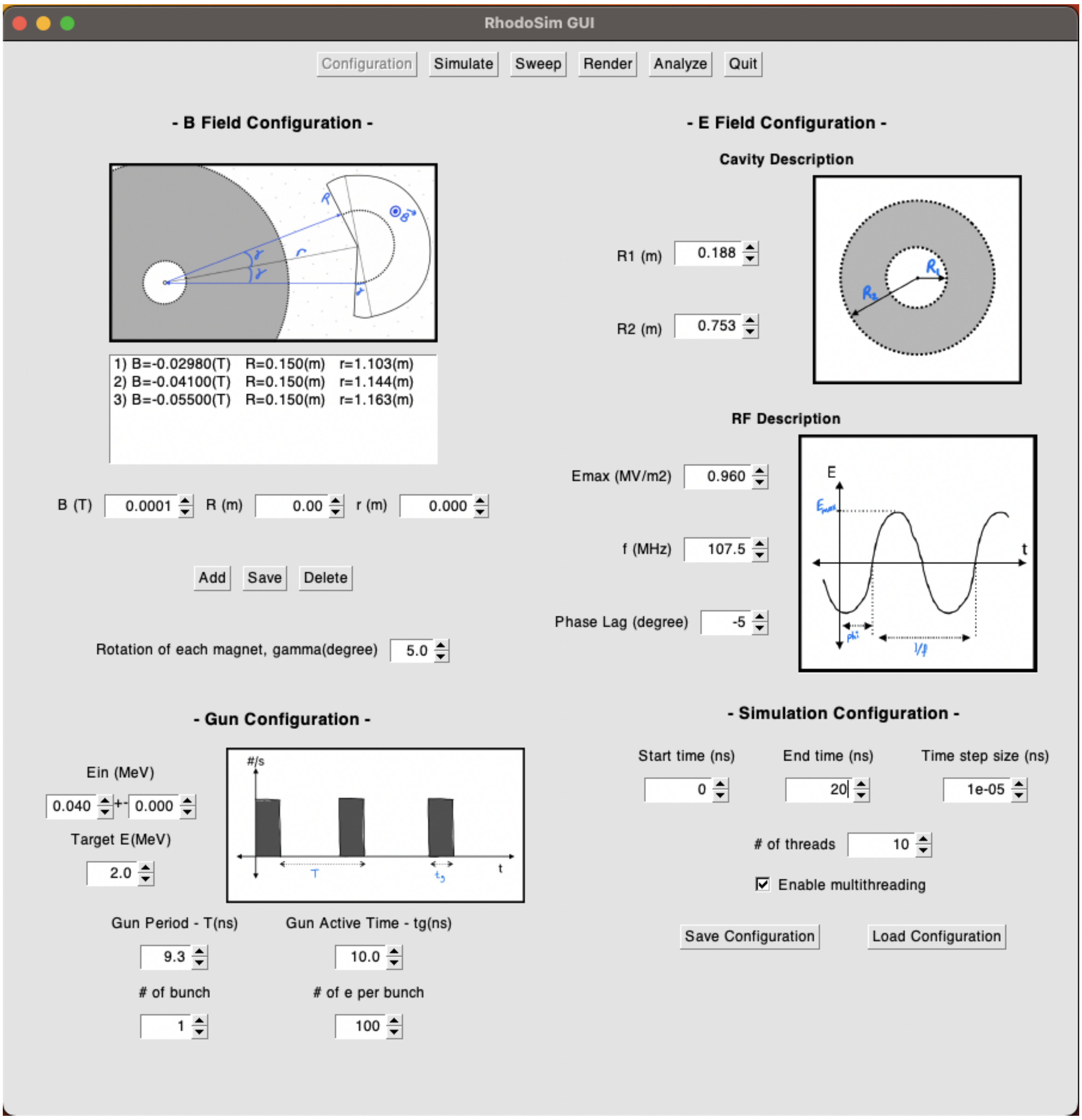


Figure 12: The configuration frame of implementated first GUI design.

has the following frames;

Configuration Frame

This frame provides an interface for specifying, saving or loading a configuration, consists of $\vec{\mathbf{B}}$ field, $\vec{\mathbf{E}}$ field, *RF* description, $\mathbf{e}^-$-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 and manages the *simulation engine*, configures and starts the simulation. It has a progress bar that shows the current progress of the simulation, communucating 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 **(**E)(t) graph of each electron are implementated 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

Parameter sweep method that was discussed in *sections 0.1.1 and 0.1.2* was integrated into *GUI* in a seperate frame named *Sweep Frame*. ϕ_{lag} sweep was the first parameter to be implemented. It takes the range of sweep, draws $\phi_{lag} vs \mu E$, $\phi_{lag} vs \sigma E$ and $\phi_{lag} vs \sigma R$ mentioned in *section 0.1.2*. This enables user with an already built accelerator to optimize $\mathbf{e}^-$-gun parameters quickly. In the following *figures 17, 18, 19 and 20*, this frame can be observed in detail.

As mentioned before, *GUI* is the latest and ongoing development effort in this project. New analysis and sweep features will be implemented and current capabilities will be improved in the near future.

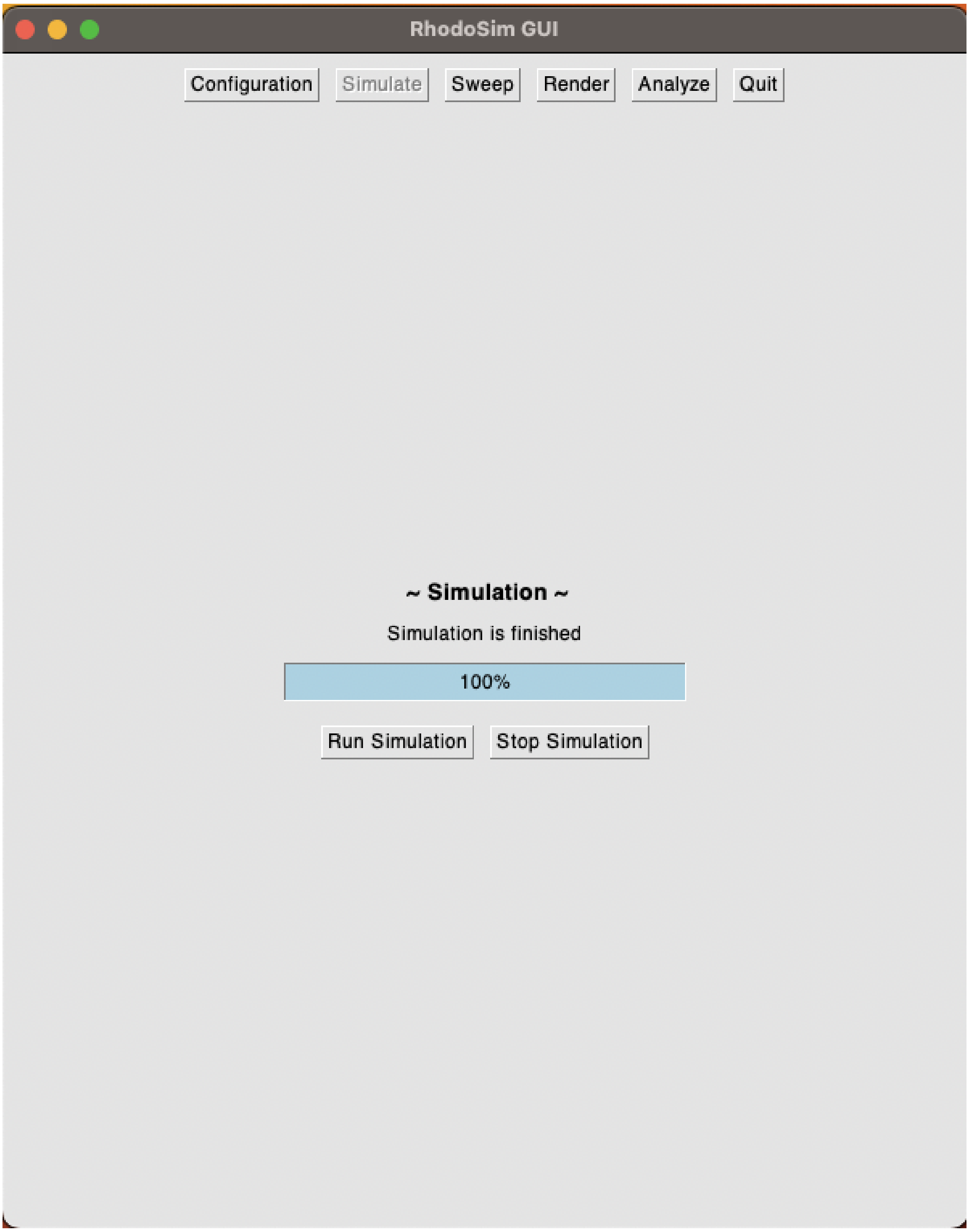
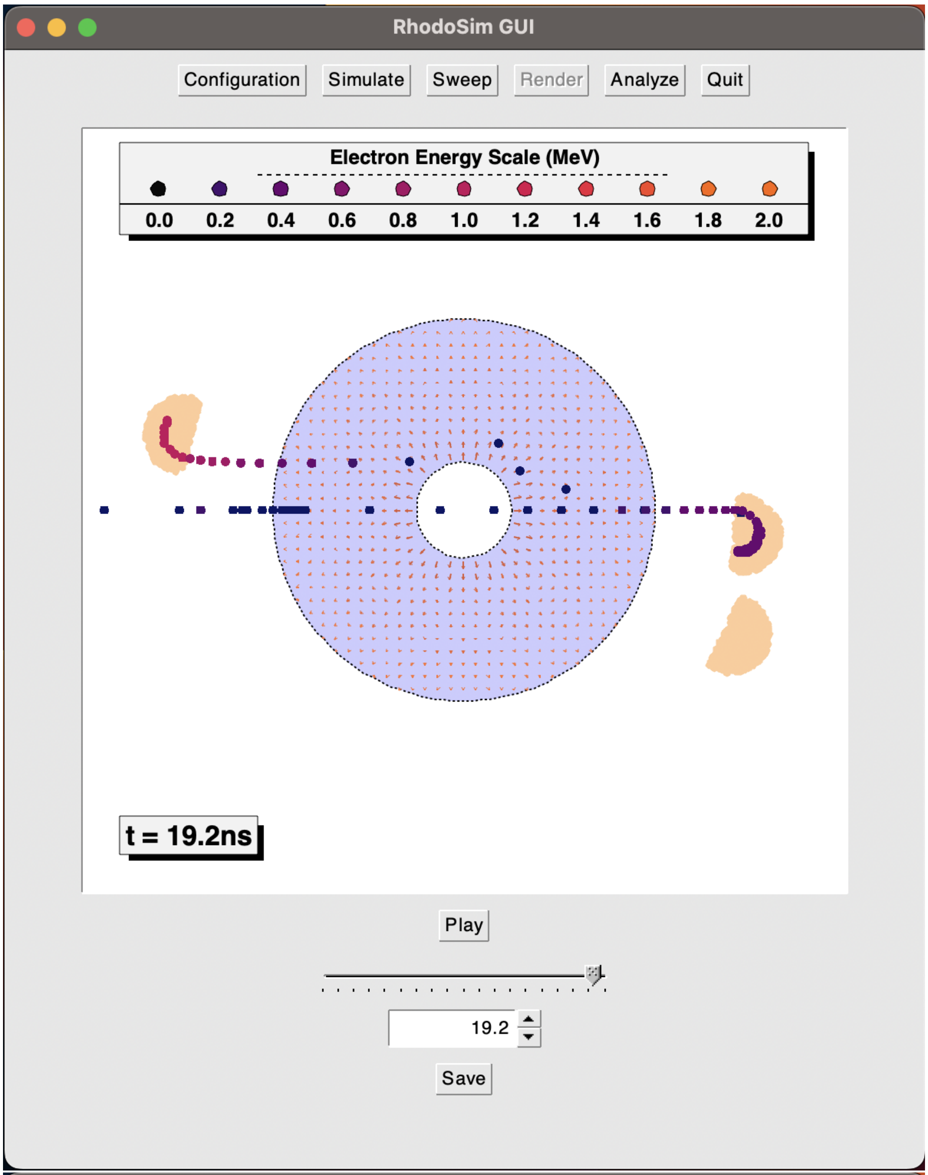
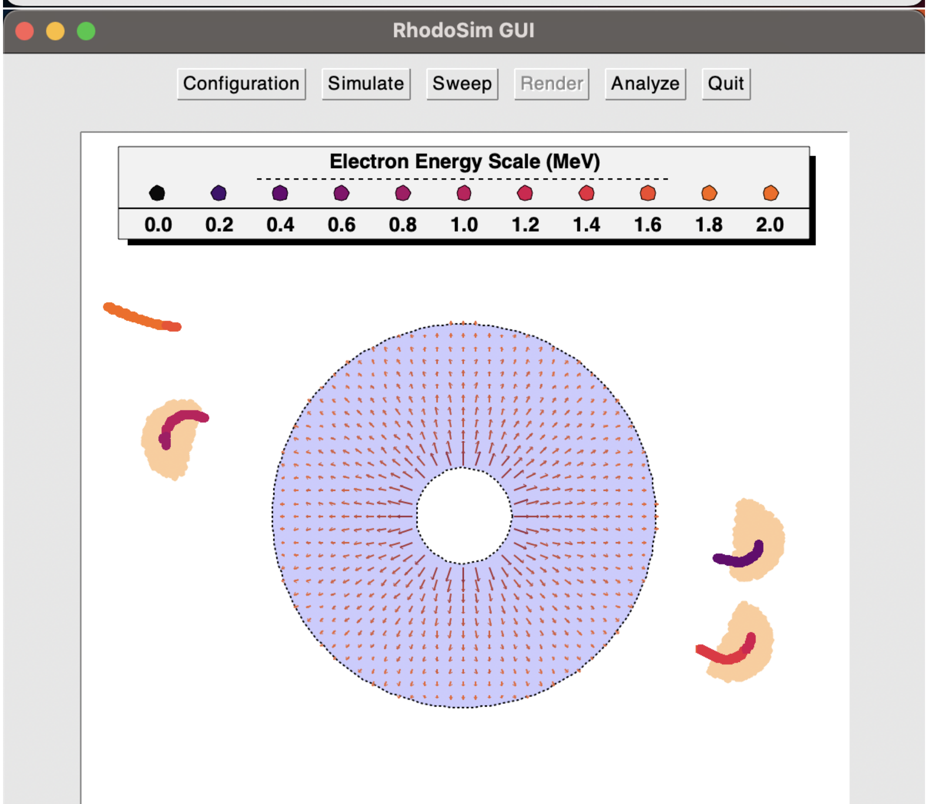


Figure 13: Simulate frame of *GUI*.


RhodoSim GUI
Configuration
Simulate
Sweep
Render
Analyze
Quit
Electron Energy Scale (MeV)
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
t = 19.2ns
Play
19.2
Save


RhodoSim GUI
Configuration
Simulate
Sweep
Render
Analyze
Quit
Electron Energy Scale (MeV)
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0

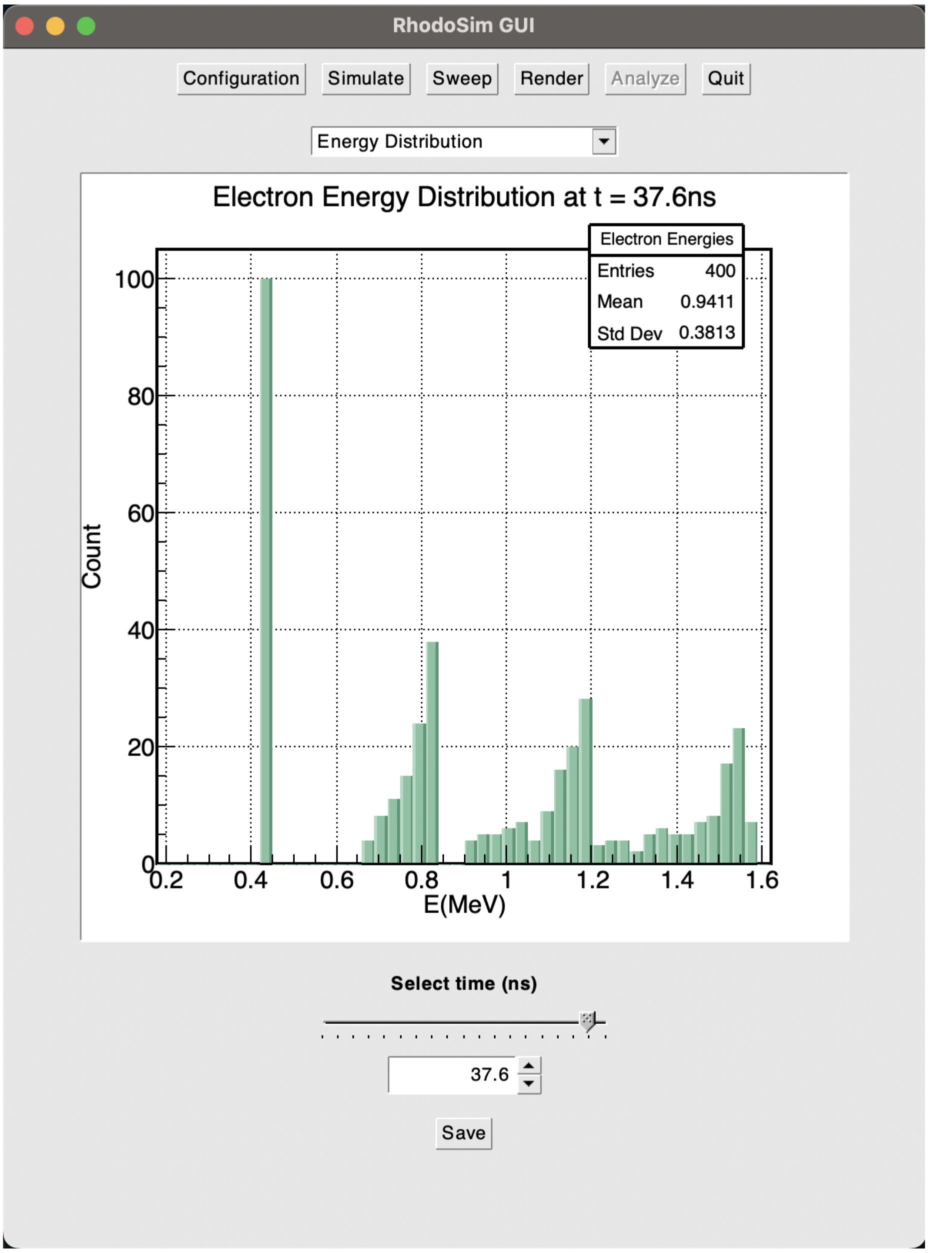
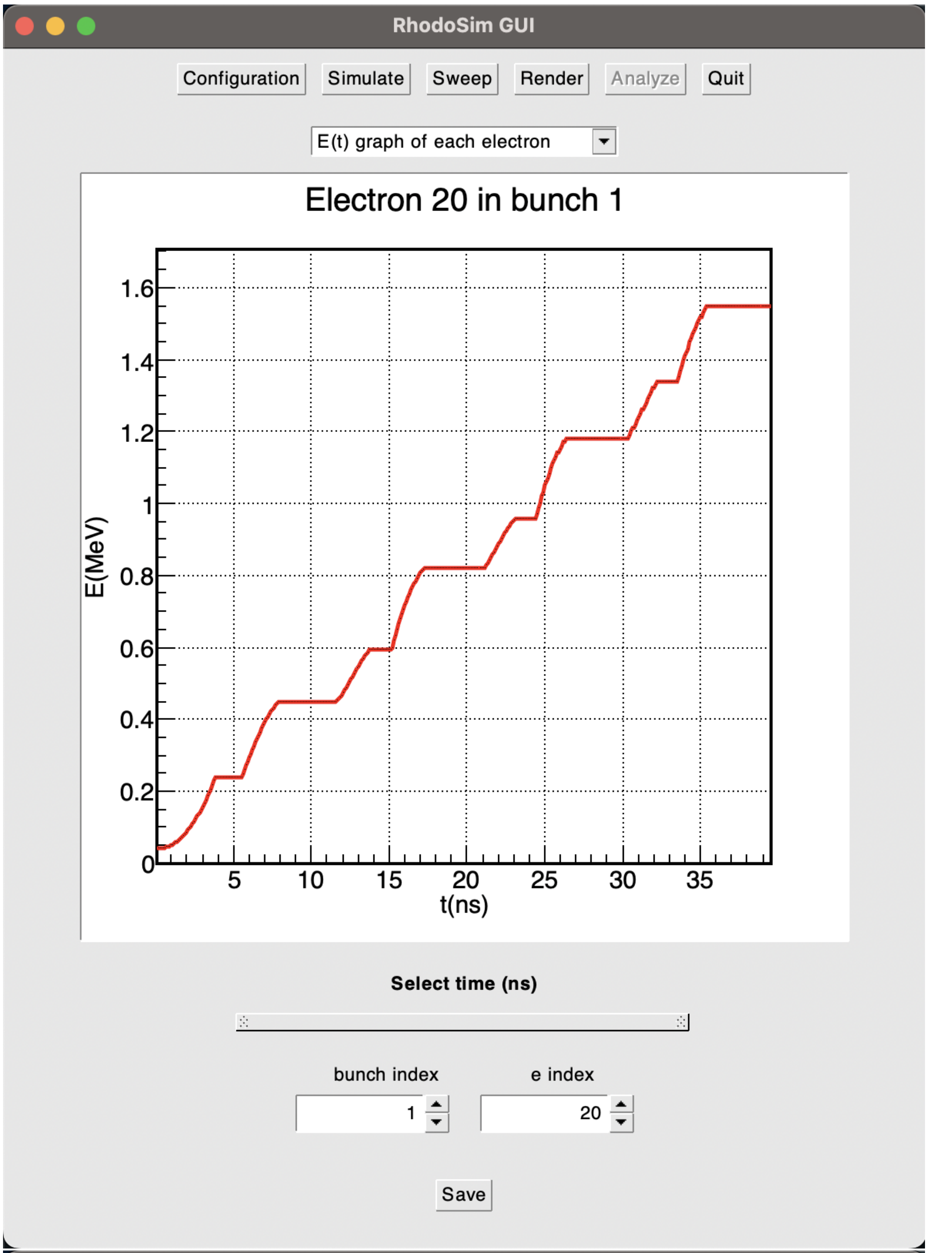
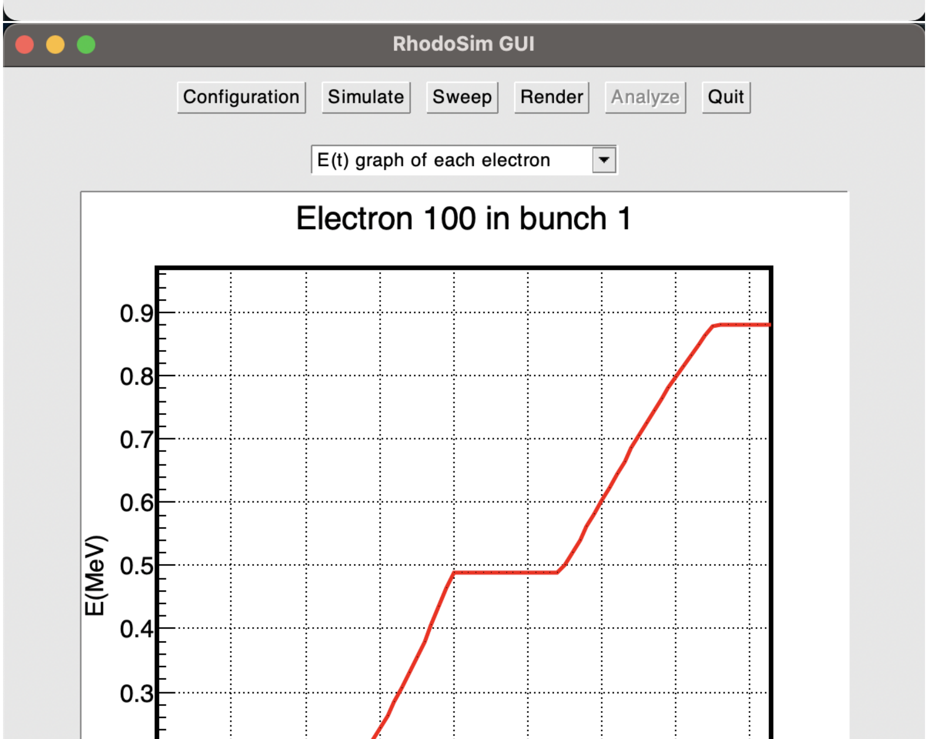


Figure 15: Analyze frame of *GUI* E distribution.


RhodoSim GUI
Configuration
Simulate
Sweep
Render
Analyze
Quit
E(t) graph of each electron
Electron 20 in bunch 1
E(MeV)
t(ns)
Select time (ns)
bunch index
e index
Save


RhodoSim GUI
Configuration
Simulate
Sweep
Render
Analyze
Quit
E(t) graph of each electron
Electron 100 in bunch 1
E(MeV)

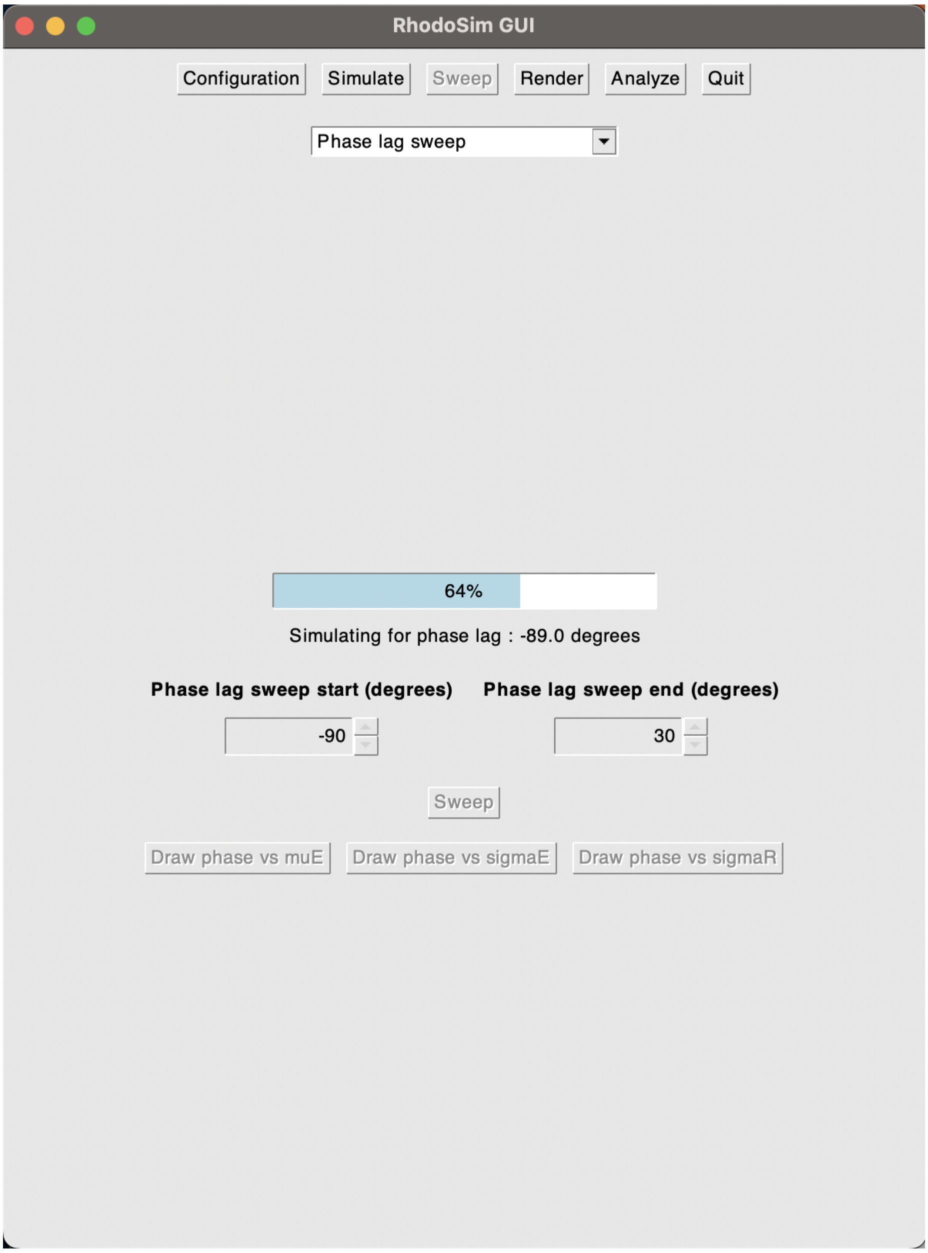


Figure 17: Sweep frame of *GUI*
ϕ_{lag} sweep running.

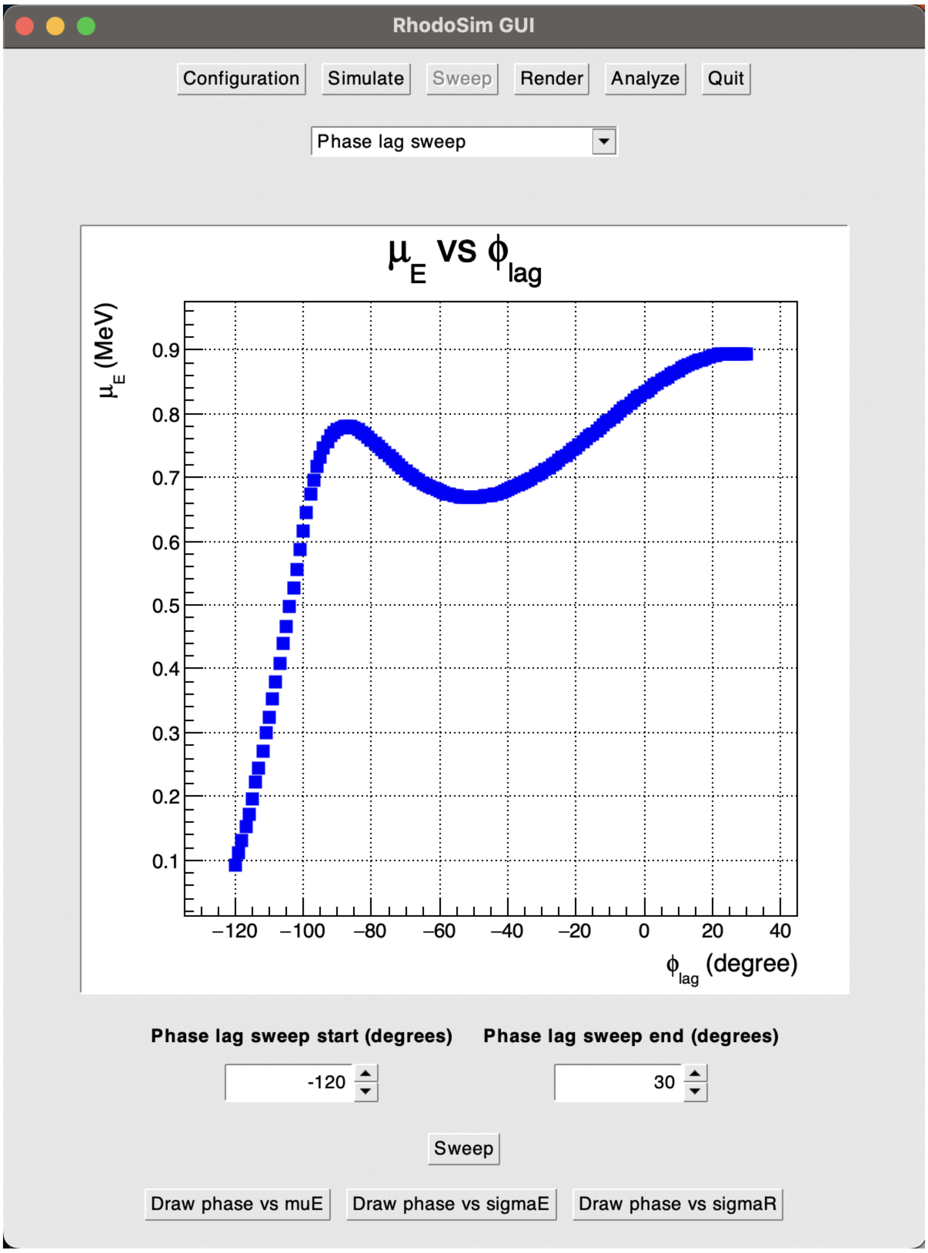


Figure 18: Sweep frame of *GUI*
ϕ_{lag} μE result.

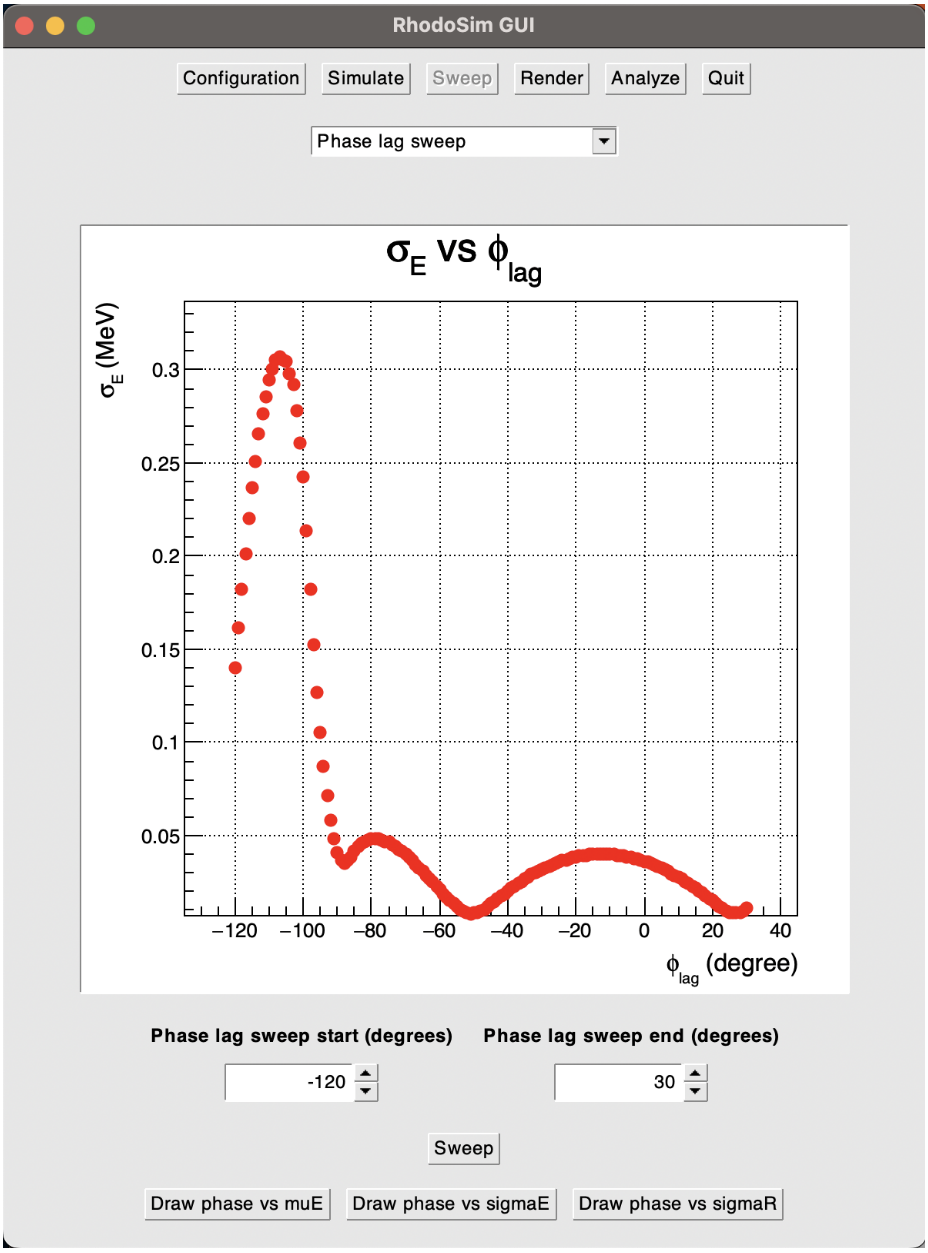


Figure 19: Sweep frame of *GUI*
ϕ_{lag} sweep σE result.

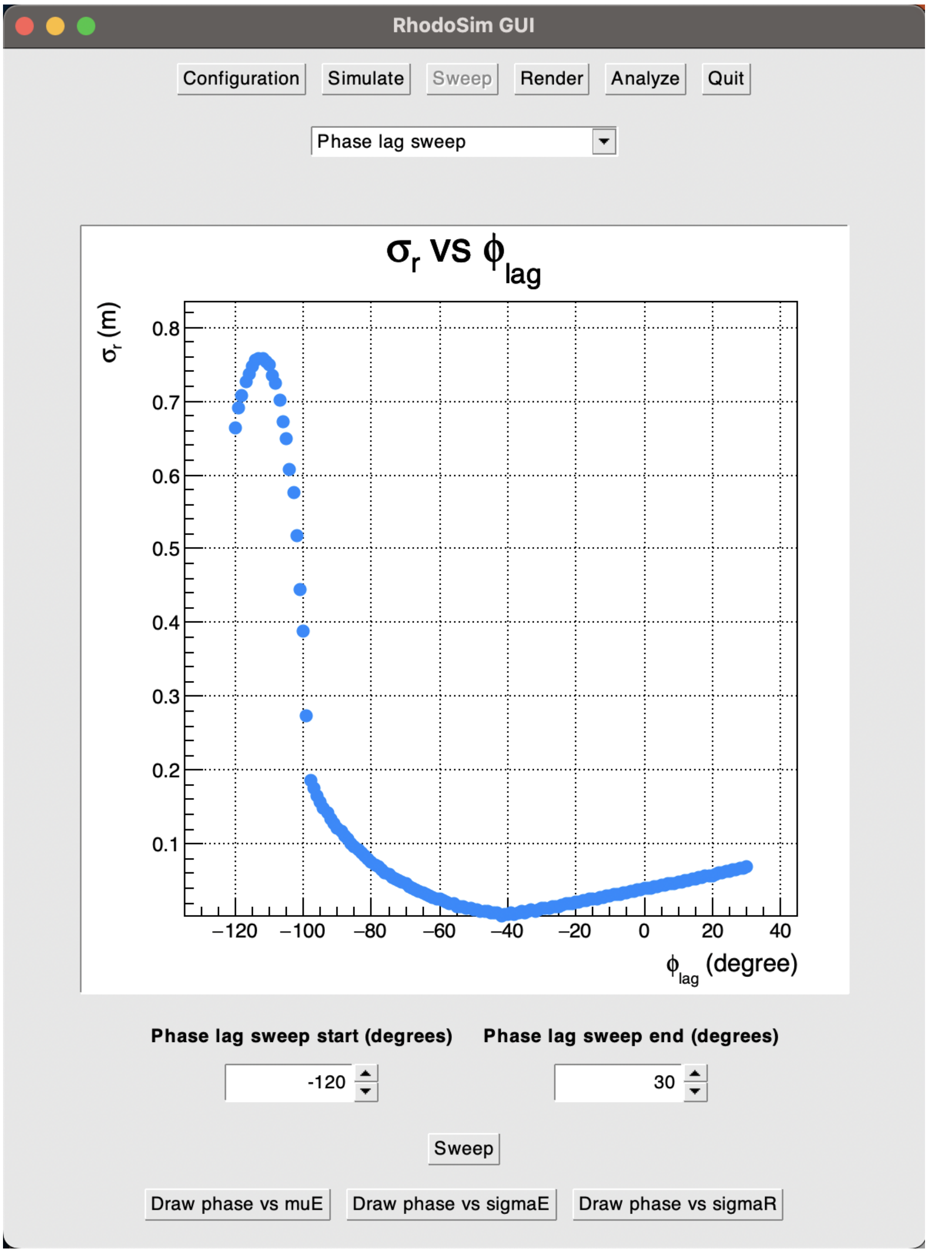


Figure 20: Sweep frame of *GUI*
ϕ_{lag} sweep σR result.

Appendix A

Intermediate Codes

```
for(double i = 2; i < 9; i += dT_out){
    t_dum += i;
    double enow = gecis(r_pos, vel, Et, t_dum);
    if( enow > maxE ){
      maxE = enow;
      t_opt = i;
    }
    t_dum = t;
}
```

```
double gecis(double r_pos, double vel, double Et, double &t){
    for(; r_pos >= -R2 && r_pos <= R2 ; t+=dT){
        vel = c*sqrt(Et*Et-E0*E0)/Et;
        double RelBeta  = vel/c;
        double RelGamma = 1.0 / sqrt(1.0-RelBeta*RelBeta);

        double ef=Eradial(r_pos*1000,t,RFphase*deg_to_rad);

        double acc=ef*1E6*eQMratio/(RelGamma*RelGamma*RelGamma);

        r_pos = r_pos + vel * dT*ns + 1/2*acc*(dT*ns)*(dT*ns);
        vel=vel+acc*dT*ns;
        RelBeta  = vel/c;
        RelGamma = 1.0 / sqrt(1.0-RelBeta*RelBeta);
        Et=RelGamma*E0;
    }
    return Et;
}
```

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

```
int phase_opt(const vector<double>& Louts, int phase_sweep_range){
    double minrms = 1;
    int opt_phase;
    for(int RFphase = -phase_sweep_range; RFphase <= phase_sweep_range; RFphase++){
        Bunch bunch1(RFphase);
        double t1 = 0;
        bunch1.bunch_gecis_t(t1);
        bunch1.reset_pos();

        for(int i = 0; i < Louts.size(); i++){
            bunch1.bunch_gecis_d(Louts[i]);
            bunch1.reset_pos();
        }

        if( bunch1.E_rms() < minrms ){
            minrms = bunch1.E_rms();
            opt_phase = RFphase;
        }
    }
    return opt_phase;
}
```

Figure A.2: ϕ_{lag} Optimization For Initial Bunch Design

```
double vector3d::operator* (const vector3d& other){
    double dot = 0;
    dot += this->x * other.x;
    dot += this->y * other.y;
    dot += this->z * other.z;
    return dot;
}
```

```
vector3d vector3d::operator% (const vector3d& other){
    double x_ = (this->y * other.z) - (this->z * other.y);
    double y_ = (this->z * other.x) - (this->x * other.z);
    double z_ = (this->x * other.y) - (this->y * other.x);
    vector3d crossed(x_, y_, z_);
    return crossed;
}
```

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

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

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

```
vector3d Electron2D::interactB_RK(const MagneticField& B, double time_interval){
    if (B.isInside(pos) == -1){
        return vector3d(0,0,0);
    }
    Electron2D e_dummy;
    e_dummy.Et = Et;
    e_dummy.pos = pos;
    e_dummy.vel = vel;
    double time_halved = time_interval*0.5;
    // get k1
    vector3d F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
    vector3d k1 = F_m * e_dummy.gamma_inv();
    // get k2
    e_dummy.move(time_halved);
    e_dummy.accelerate(k1, time_halved);
    F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
    vector3d k2 = F_m * e_dummy.gamma_inv();
    // get k3
    e_dummy.vel = vel;
    e_dummy.accelerate(k2, time_halved);
    vector3d k3 = F_m * e_dummy.gamma_inv();
    // get k4
    e_dummy.vel = vel;
    e_dummy.move(time_halved);
    e_dummy.accelerate(k3, time_interval);
    F_m = (e_dummy.vel % B.getField(pos))*eQMratio;
    vector3d k4 = F_m * e_dummy.gamma_inv();

    return (k1 + k2*2 + k3*2 + k4)/6;
}
```

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

```
void Electron2D::interactRK_ActorE(const RFField& E, const MagneticField& B, double time_interval){
    vector3d run_kut_E = interactE_RK(E, time_interval);
    vector3d run_kut_B = interactB_RK(B, time_interval);

    vector3d acc = run_kut_E + run_kut_B;

    move(acc, time_interval/2);
    accelerate(acc, time_interval);
    move(acc, time_interval/2);
}
```

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

```
    void Electron2D::interactRK(RFField& E, MagneticField& B, const double time, double time_interval){
        Electron2D e_dummy;
        e_dummy.Et = Et;
        e_dummy.pos = pos;
        e_dummy.vel = vel;

        // Calculate k1
        vector3d acc_E = E.actOn(e_dummy);
        vector3d acc_B = B.actOn(e_dummy);
        vector3d acc = acc_E + acc_B;
        e_dummy.move(time_interval);
        e_dummy.accelerate(acc, time_interval);
        vector3d pos_k1 = e_dummy.pos, vel_k1 = e_dummy.vel;

        // Calculate k2
        e_dummy.pos = (pos + pos_k1)*0.5;
        e_dummy.vel = (vel + vel_k1)*0.5;
        e_dummy.Et = e_dummy.gamma()*E0;
        E.update(time + time_interval*0.5);

        acc_E = E.actOn(e_dummy);
        acc_B = B.actOn(e_dummy);
        acc = acc_E + acc_B;
        e_dummy.move(time_interval);
        e_dummy.accelerate(acc, time_interval);
        vector3d pos_k2 = e_dummy.pos, vel_k2 = e_dummy.vel;

        // Calculate k3
        e_dummy.pos = (pos + pos_k2)*0.5;
        e_dummy.vel = (vel + vel_k2)*0.5;
        e_dummy.Et = e_dummy.gamma()*E0;
        E.update(time + time_interval*0.5);

        acc_E = E.actOn(e_dummy);
        acc_B = B.actOn(e_dummy);
        acc = acc_E + acc_B;
        e_dummy.move(time_interval);
        e_dummy.accelerate(acc, time_interval);
        vector3d pos_k3 = e_dummy.pos, vel_k3 = e_dummy.vel;

        // Calculate k4
        E.update(time + time_interval);

        acc_E = E.actOn(e_dummy);
        acc_B = B.actOn(e_dummy);
        acc = acc_E + acc_B;
        e_dummy.move(time_interval);
        e_dummy.accelerate(acc, time_interval);
        vector3d pos_k4 = e_dummy.pos, vel_k4 = e_dummy.vel;

        E.update(time);
        pos = (pos_k1 + pos_k2*2 + pos_k3*2 + pos_k4)/6;
        vel = (vel_k1 + vel_k2*2 + vel_k3*2 + vel_k4)/6;
        Et = gamma()*E0;
}
```

Figure A.7: RK4-2 implemenation of $\mathbf{e}^-$ - **EM**

```
void RhodotronSimulator::_runMT(){
    gun.fireAllWithFireTimesMT();

    MTEngine.setupPool(time_interval, start_time, end_time, gun, E_field, B_field, gun.thread_bunchs);

    STEPS_TAKEN = 0;
    simulation_time = start_time;
    while (simulation_time < end_time + time_interval ){
        if (STEPS_TAKEN % log_interval() == 0){
            E_field.update(simulation_time);
            logEfield(simulation_time, simulation_time + time_interval > end_time);
            notifyUI(MTEngine.getAverageTime());
        }
        simulation_time+=time_interval;
        STEPS_TAKEN++;
    }
    bool end = false;
    while (!end){
        double time = MTEngine.getAverageTime();
        notifyUI(time);
        if ( time >= end_time ){
            end = true;
        }
        this_thread::yield();
    }
    MTEngine.join();

}
```

Figure A.8: Multithreading main-thread logic.

```
void Gun::fireAllWithFireTimesMT(){
    std::random_device rd;
    std::mt19937 e2(rd());
    std::normal_distribution<double> Edist(Ein, sEin);

    for(_fired_bunch= 0; _fired_bunch < bunch_count; _fired_bunch++){
        for(_fired_e_in_current_bunch= 0; _fired_e_in_current_bunch < e_per_bunch; _fired_e_in_current_bunch++){

            double E = (sEin == 0 ) ? Ein : Edist(e2);

            double fire_time = (ns_between_each_electron_fire * _fired_e_in_current_bunch) + _fired_bunch*gun_period;

            auto burrowed_e = bunchs[_fired_bunch].AddElectronGiveAddress(E, gunpos, gundir, fire_time);

            int thread_index = (_fired_e_in_current_bunch + _fired_bunch*e_per_bunch)%thread_bunchs.size();

            thread_bunchs[thread_index]->push_back(burrowed_e);
        }
    }
}
```

Figure A.9: Multithreading electron assign logic.

```
void MultiThreadEngine::setupPool( double _time_interval, double _start_time, double _end_time, Gun& gun,
    CoaxialRFField& RF, MagneticField& B, vector<shared_ptr<vector<shared_ptr<Electron2D>>>>& e_list){
    threads.reserve(thread_count);

    for(int i = 0; i < thread_count && i == threads.size(); i++){

        child_notifier_mutexes.push_back(make_shared<mutex>());
        child_times.push_back(make_shared<double>());

        auto _E = RF.Copy();
        auto _B = B.LightWeightCopy();
        double time_between_fires = thread_count*gun.getGunActiveTime()/gun.getElectronsPerBunch();
        double first_fire_time = i*gun.getGunActiveTime()/gun.getElectronsPerBunch();
        ThreadArguments thread_arguments(i, _time_interval, _start_time, _end_time, &gun, _E, _B, e_list[i], first_fire_time, time_between_fires);
        thread_arguments.parent_notifier_mutex = child_notifier_mutexes[i];
        thread_arguments.current_thread_time = child_times[i];
        threads.push_back(thread(threadLoop, thread_arguments));
    }
}
```

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

```
void threadLoop(ThreadArguments thread_arguments){
    uint64_t count = 0;

    double sim_time = thread_arguments.start_time;

    while(sim_time < thread_arguments.end_time + thread_arguments.time_interval){
        thread_arguments.E->update(sim_time);
        thread_arguments.i_args.time = sim_time;

        if ( count % (unsigned long)(0.1/thread_arguments.time_interval) == 0){
            saveElectronInfoForSingleThread(thread_arguments.i_args);
            // Notifiy the main thread
            if(thread_arguments.parent_notifier_mutex->try_lock()){
                *thread_arguments.current_thread_time = sim_time;
                thread_arguments.parent_notifier_mutex->unlock();
            }
        }

        interactForSingleThread(thread_arguments.i_args);
        // save electron info here
        sim_time+= thread_arguments.time_interval;
        count++;
    }
    thread_arguments.parent_notifier_mutex->lock();
    *thread_arguments.current_thread_time = sim_time;
    thread_arguments.parent_notifier_mutex->unlock();
}
```

Figure A.11: Multithreading worker-thread logic.

```cpp
void UIThreadWork(UIThreadArgs args){
    int UI_WORK_PIECE = SIM_WORK_MASK;
    if ( !args.isService ) {
        UI_WORK_PIECE = 50;
    }

    double piece = (args.end_time - args.start_time)/UI_WORK_PIECE;

    if ( !args.isService ) {
        std::string sim_running_msg = "...Simulation is running...";
        for(int i = 0; i < 26 - sim_running_msg.size()/2 ; i++){
            std::cout << " ";
        }
        std::cout << sim_running_msg <<"\n";
    }

    args.ui_mutex->lock();
    double simtime = *(args.simulation_time);
    args.ui_mutex->unlock();

    if ( !args.isService ) {
        std::cout << "V";
        for(int i = 0; i < 51; i++){
            std::cout << "_";
        }
        std::cout << "V\n[" << std::flush;
    }

    int count = 0;
    bool running = true;
    while(running && (simtime < args.end_time || count < UI_WORK_PIECE )){
        if( simtime > count * piece ){
            if ( !args.isService ) {
                std::cout << "#" << std::flush;
            }
            count++;
            args.state_mutex->lock();
            running = *args.state_ptr & SIM_RUNNING;
            *args.state_ptr = (*args.state_ptr & ~SIM_WORK_MASK) | (count & SIM_WORK_MASK);
            if (args.isService) write(args._fd, args.state_ptr, SIGNAL_SIZE);
            args.state_mutex->unlock();
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(25));
        args.ui_mutex->lock();
        simtime = *(args.simulation_time);
        args.ui_mutex->unlock();

    }
    if ( !args.isService ) {
        std::cout << "#]\n\n" << std::flush;
        std::cout << "     ...Simulation is finished successfully...\n\n" << std::flush;
    }

    args.state_mutex->lock();
    *args.state_ptr |= SIM_RENDERING;
    if ( args.isService ) write(args._fd, args.state_ptr, SIGNAL_SIZE);
    args.state_mutex->unlock();
}
```

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

Appendix B

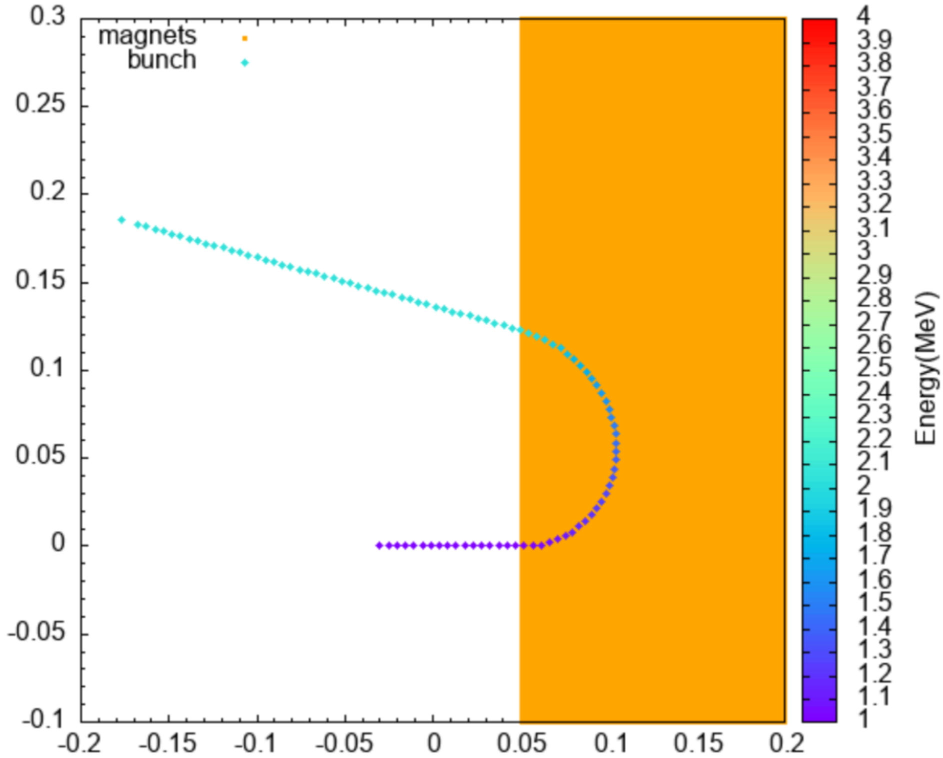
Example Simulation Runs

```
Optimal phase with the least RMS : -5

Simulation settings :
ph = -5 deg, gt = 1 ns, enum = 1000
dT = 0.001 ns, dT_out = 0.01 ns

For the 1th magnet:
Optimum out path = 0.81044 m
Magnet guide = 0.25852 m
Rho = 0.088477 m
Drift time of the first electron in the bunch : 7.688 ns
Drift time of the last electron in the bunch : 7.487 ns
Max energy = 0.47581 MeV
RMS = 0.0058165 MeV

For the 2th magnet:
Optimum out path = 1.0833 m
Magnet guide = 0.37766 m
Rho = 0.098898 m
Drift time of the first electron in the bunch : 5.597 ns
Drift time of the last electron in the bunch : 5.617 ns
Max energy = 0.89172 MeV
RMS = 0.0099018 MeV

For the 3th magnet:
Optimum out path = 1.1705 m
Magnet guide = 0.41573 m
Rho = 0.10223 m
Drift time of the first electron in the bunch : 5.314 ns
Drift time of the last electron in the bunch : 5.325 ns
Max energy = 1.298 MeV
RMS = 0.013879 MeV

Electron with the most energy : 623) 1.6999 MeV,        RMS of bunch : 0.017981 MeV

Total steps calculated : 12468052652
Simulation finished in : 632050015 us     ( 632.1 s )

```

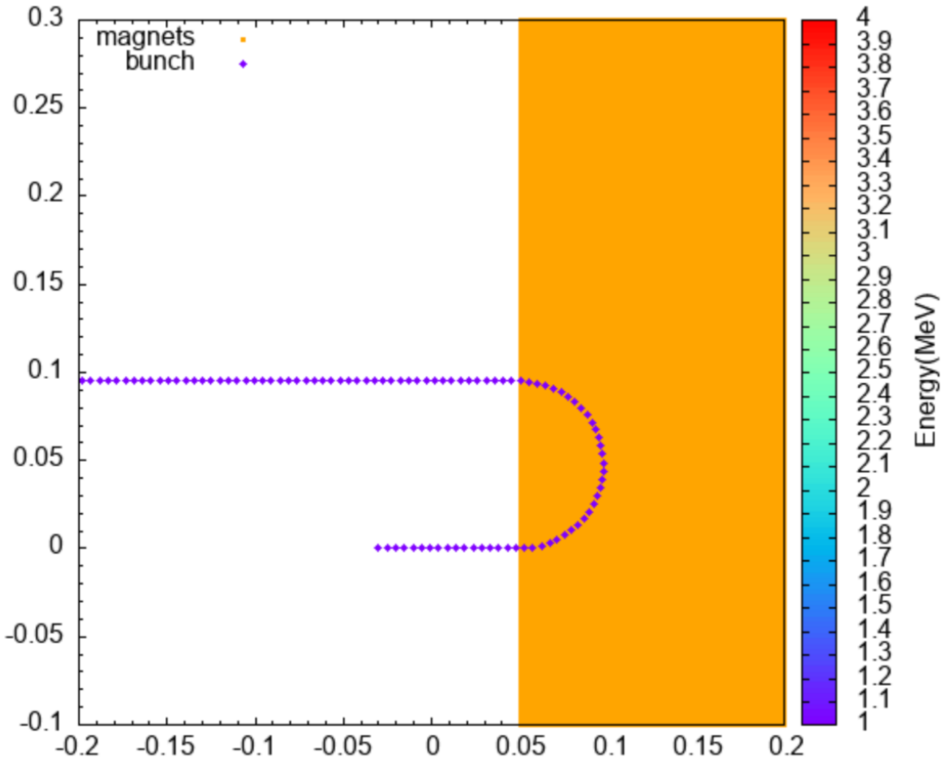
Figure B.1: ϕ_{lag}, ρ & L optimization at
$P = 12\text{KW}$, $R_1 = 0.188\text{m}$, $R_2 = 0.753\text{m}$, $t_g = 1\text{ns}$, $E_{in} = 40\text{KeV}$

```
Optimal phase with the least RMS : 0

Simulation settings :
ph = 0 deg, gt = 0.8 ns, enum = 1000
dT = 0.001 ns, dT_out = 0.01 ns

For the 1th magnet:
Optimum out path = 0.80787 m
Magnet guide = 0.2574 m
Rho = 0.088379 m
Drift time of the first electron in the bunch : 7.629 ns
Drift time of the last electron in the bunch : 7.48 ns
Max energy = 0.47579 MeV
RMS = 0.0038689 MeV

For the 2th magnet:
Optimum out path = 1.0833 m
Magnet guide = 0.37765 m
Rho = 0.098898 m
Drift time of the first electron in the bunch : 5.589 ns
Drift time of the last electron in the bunch : 5.605 ns
Max energy = 0.89169 MeV
RMS = 0.0068848 MeV

For the 3th magnet:
Optimum out path = 1.1705 m
Magnet guide = 0.41573 m
Rho = 0.10223 m
Drift time of the first electron in the bunch : 5.311 ns
Drift time of the last electron in the bunch : 5.318 ns
Max energy = 1.298 MeV
RMS = 0.0096887 MeV

Electron with the most energy : 629) 1.6999 MeV,        RMS of bunch : 0.012318 MeV

Total steps calculated : 12455378454
Simulation finished in : 631136046 us     ( 631.1 s )
```

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



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



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

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

```
#            Rhodotron Simulation Configuration File
# ================================================================
#    M.Furkan Er                                            22/09/2022
# ================================================================
#
# emax = Maximum electric field strength (MV/m)
# ein = Energy of electrons coming out of the gun (MeV)
# einstd = Standard deviation of energy of electrons coming out of the gun (MeV)
# targeten = Max energy on the output gif (MeV)
# freq = Frequency of the RF field (MHz)
# phaselag = phase lag of the first electrons (degree)
# starttime = time to start firing the gun (ns)
# endtime = ns to run the simulation (ns)
# dt = time interval to do the calculations (ns)
# guntime = how long a gun pulse is (ns)
# gunperiod = time between two gun pulses (ns)
# enum = number of electrons to simulate in a bunch
# bunchnum = number of times the gun fires
# r1 = radius of the inner cylinder (m)
# r2 = radius of the outer cylinder (m)
# epath = path to store the electric field data
# bpath = path to store the magnetic field data
# cpath = path to store the settings
# ppath = path to store electron data
# multh = enable or disable multitheading
# thcount = set the maximum thread to be used
# magrotation = degrees of rotation to enter each magnet
# addmagnet = takes 3 input. (B , R, < Radial distance of center >)
# output = output file name


# E FIELD CONFIGURATION
emax=1.170
freq=107.3
phaselag=10.0
r1=0.1840
r2=0.7380

# B FIELD CONFIGURATION
magrotation=5.0

# GUN CONFIGURATION
einmean=0.040
einstd=0.0000
targeten=2.0
guntime=1.0
gunperiod=9.3
enum=50
bunchnum=1

# SIM CONFIGURATION
starttime=0
endtime=10
dt=0.0000010000
epath=xy/rf.dat
bpath=xy/magnet.dat
cpath=xy/settings.dat
ppath=xy/paths/
multh=1
thcount=10
```

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

```
-- Simulation Configuration --
Emax : 0.96     MV/m
Freq : 107.5    MHz
Phase Lag : -5  degree
EndTime : 100   ns
dT : 0.0001     ns
guntime : 0.8   ns
gunperiod : 9.3 ns
enum : 100
bunchnum : 2
R1 : 0.188241   m
R2 : 0.752967   m
Magnet count :  5
Ein : 0.04      MeV
TargetE : 2.5   MeV
--------------------------------

             ...Simulation is running...
V_____________________________________________________V
[####################################                 ]
```

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

Appendix C

Data and Graphs

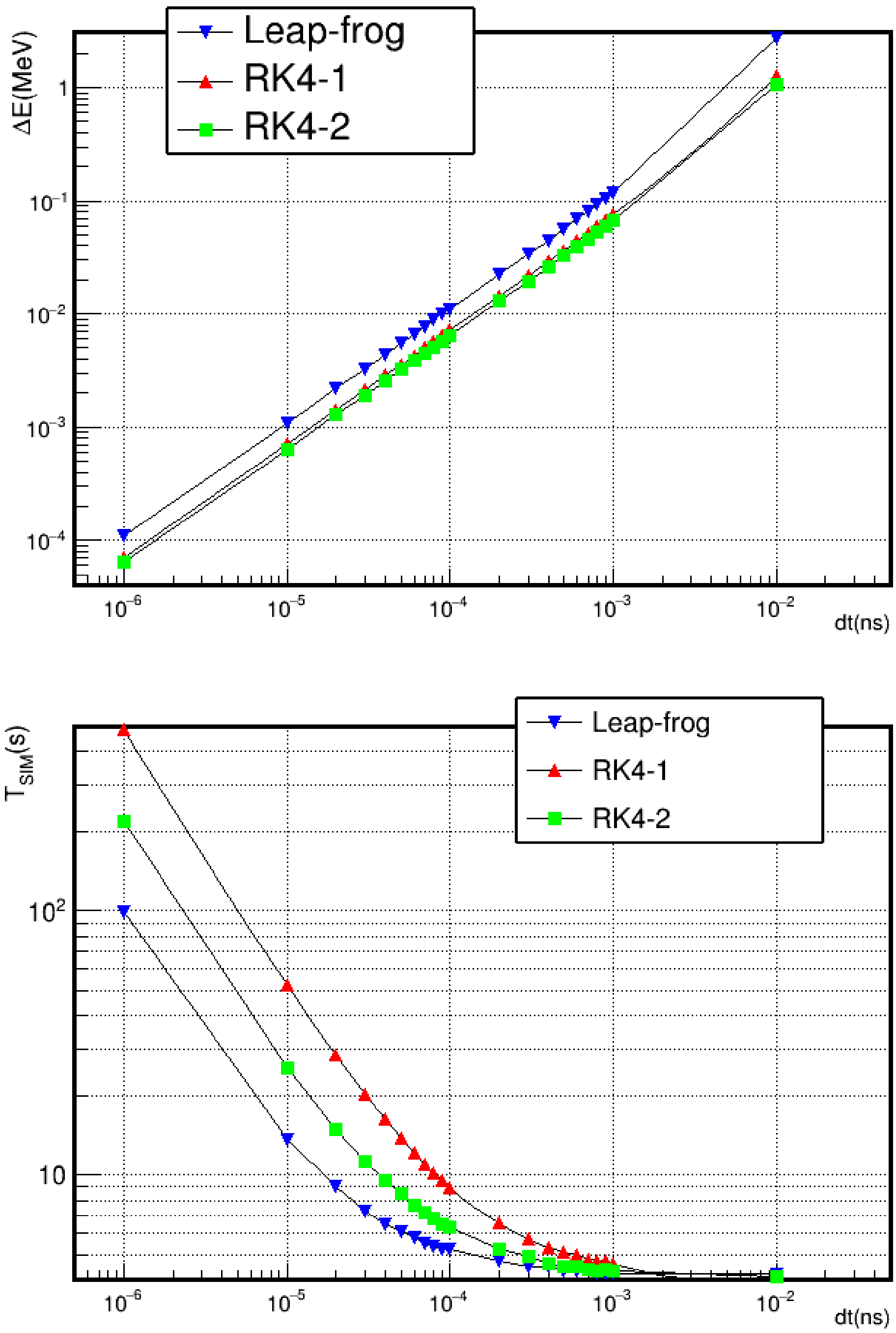


Figure C.1: Comparing Leap-frog, RK4-1, RK4-2 performance on $e^- - \vec{B}$ interaction

dt(ns)	ΔE_{avg}(MeV)	μT_{sim}(s)	σT_{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
$E_{in} = 1$MeV, $\mathbf{B}$=0.1T, $t_{end} = 5$ns

dt(ns)	ΔE_{avg}(MeV)	μT_{sim}(s)	σT_{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
$E_{in} = 1$MeV, $\mathbf{B}$=0.1T, $t_{end} = 5$ns