university of groningen / faculty of science and engineering / mathematics and applied mathematics

Stability of Simulated Decelerating Cold Particles in a Stark Decelerator

Bachelor's Project Mathematics and Physics

July 2020

Student: L. Wester

First supervisor: Prof.dr. S. Hoekstra

Second assessor: Dr. A.E. Sterk

Abstract

To either advance our confidence in the *Standard Model*, or to invite new extensions, it is interesting to measure the value of the electron electric dipole moment (*e-EDM*). The standard model suggests this value less than 10^{-38} e cm. To measure an EDM of this magnitude, an experiment was set up. This involves decelerating and cooling heavy polar molecules. The deceleration is done using a *Stark Decelerator*. As particles enter the decelerator and slow down, a portion of this particles is lost due to them escaping the velocity traps when their kinetic energy is too high, or due to instability of the particles as they are decelerated. In this thesis, the effects of several changes to the deceleration methods are measured. This is done through simulation software rather than running the actual decelerator.

Contents

1 Introduction

The Standard Model of Particle Physics predicts a dipole moment for the electric field of an electron. The order of magnitude of the predicted field is around 10^{-38} e cm. If this moment were measured, it would contribute significantly to the confidence in the Standard Model. Showing that the electron electric dipole moment (electron-EDM) is greater than this predicted value would invite the postulation of new physics [5].

To investigate the electron-EDM, the University of Groningen has launched the electron-EDM experiment. In this experiment, a stream of barium fluoride (BaF) molecules with a strong dipole moment is produced. This stream of molecules is placed inside a sinusoidal electric field. In the presence of a strong electric field the BaF molecules exhibit *Stark Splitting*, where the spectral lines of the molecule split due to the *Stark effect*. The Stark effect is the electric analogue of the magnetic Zeeman effect, where the electric field pulls on the nucleus and pushes on the electron, causing a dipole moment. Then the molecules shift to the minima of the sine due to the gradient in the field. This creates pockets of molecules sitting at the bottom of a potential well. Then, the frequency of the sine is decreased along the length of the decelerator, causing the molecule pockets to slow down, which in turn improves the accuracy of the detector.

We are dealing with molecules sitting in a 3D potential well and their speed relative to this well is nonzero. Therefore, the resulting equations of motion are fairly complicated and not analytically solvable. The stability of the molecules in this system is thus not easily verified. To test the stability of the particles in the decelerator we are restricted to either running the experiment many times, or to use simulation software. The parameters that are experimented with are not always easy to produce in real life. It is therefore worthwhile to run simulations before running the actual experiment in order to get an idea of what changes have a positive effect on the accuracy of the experiment before investing significant time and effort into producing these conditions in the actual experiment.

In this research it is postulated that there exist shapes for the electric field that yield a more stable system than a pure sine. This gives the research question for this thesis: What shape of waveform yields the most particles at the end of the decelerator, and which numerical methods can best be used to determine this?

2 Setup

To determine the electron EDM, a setup was created at the University of Groningen [3]. This setup consists of several components, one of which is a narrow tube made of small metal rings, placed at 6 mm intervals. A schematic of these rings is shown in Figure 1. Each of these rings is hooked up to a high-voltage supply,

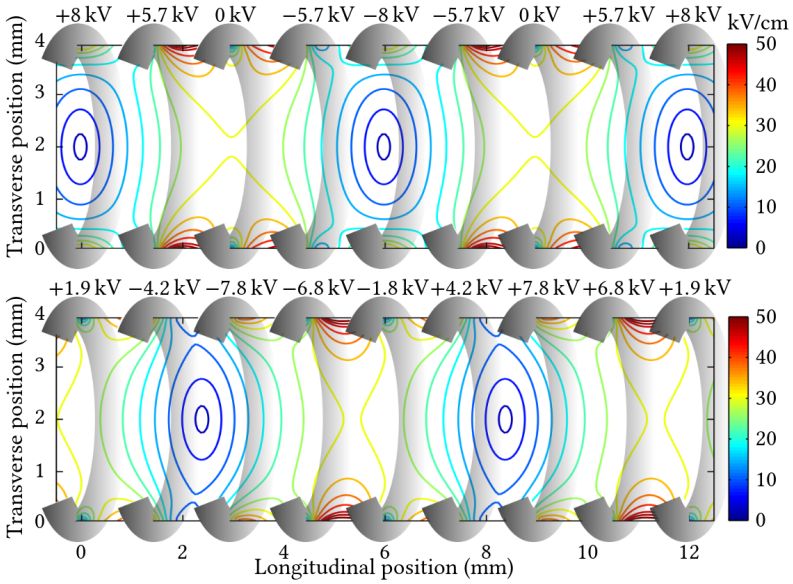

Figure 1: A schematic of the Stark decelerator, showing the electrically charged rings and their potentials. The electric field is shown inside the rings. The rings are showed twice at a different moment in time to illustrate the movement of the electric field inside the rings.

whose voltages are configured to vary with time. The voltages are then varied in such a way that the total electric field of the rings moves along the direction of the decelerator. Then, as the stream of low field seeking particles is emitted through the tube, the minima of the sine form potential wells for the molecules. The oscillation frequency is configured in such a way that the speed of the molecules and the speed of the molecule traps initially coincide. As the particles travel through the tube, the oscillation frequency is gradually lowered. This reduces the velocity of the molecules in the trap, but also causes molecules to leave the trap. There are two possible ways for the molecule to leave the trap. It can leave the trap in the transverse direction, in which case it will collide with one of the rings and be lost to the experiment. The second option is for the molecule to leave the trap in the longitudinal direction. This is only possible if the particle has enough kinetic energy relative to the position of the trap to climb the potential and exceed the maximum. If this is the case, then the particle will also have enough energy to exceed the next maximum, and thus it will generate noise in the final experiment. At the end of the tube, the fraction of the molecules that remains trapped is slowed down to almost a standstill, which allows the EDM measurement to be performed.

3 Theory

In this section we will discuss the theory used throughout this thesis. This will include the underlying physical phenomena as well as the mathematics used to model to model these phenomena.

3.1 Prior art

The BaF molecules in the trap are subject to a set of equations of motion. These were derived in an earlier research by Jeroen Muller [1]:

$$\begin{aligned}
\frac{dx}{dt} &= v_x, \\
\frac{dy}{dt} &= v_y, \\
\frac{dz}{dt} &= v_z, \\
\frac{dv_x}{dt} &= -\frac{1}{m}\frac{dS}{dE}\left(|E|\left(x,y,z\right)\right)\left[\nabla|E|\left(x,y,z\right)\right]_x, \\
\frac{dv_y}{dt} &= -\frac{1}{m}\frac{dS}{dE}\left(|E|\left(x,y,z\right)\right)\left[\nabla|E|\left(x,y,z\right)\right]_y, \\
\frac{dv_z}{dt} &= -\frac{1}{m}\frac{dS}{dE}\left(|E|\left(x,y,z\right)\right)\left[\nabla|E|\left(x,y,z\right)\right]_z.
\end{aligned}$$

These equations of motion are not solvable by hand, but in the simulation they are solved using the Dormand-Prince method. They will be used throughout the simulation code to simulate the movement of the particles.

3.2 Simulation Program

The decelerator is simulated to allow for the researcher to try new ideas without having to run the full scale setup. The simulation is split in two parts, one written in Python for the frequently changing parts, and one written in C++ for the compute-heavy tasks that need more performance. The simulation splits each run into one or more *batches*. These batches contain a full set of simulation parameters and can be run in parallel, such as the speed and distribution of the incident molecules, the speed of the waveform along the rings and the number of molecules to simulate. It is possible to create a new waveform in the Python code by supplying harmonics of a sine wave, as we will do in Section 4.2. The waveform is then computed from the harmonics in the Python code. Because it is not possible to share functions between Python and C++ in a performant way, the Python constructs a *spline* of the waveform, which is then sent to C++. A spline is a set of points of evaluation of a function, that can then later be used to determine the original function. The C++ code in turn interpolates this spline when it needs to evaluate the waveform. Then the position and velocity of the

particle are evaluated by computing the electric field that the particle experiences.

3.3 Dormand-Prince

The Dormand-Prince method (DOPRI method) is a method to solve ODEs and belongs to the family of Runge-Kutta solvers. DOPRI is an iterative method, whose steps are given by the following formulae:

$$
\begin{aligned}
y_{n+1} &= y_n + h\sum_{i=1}^{7} b_i k_i, \text{ with} \\
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + 1/5h, y_n + h(1/5 \cdot k_1)), \\
k_3 &= f(t_n + 3/10h, y_n + h(3/40 \cdot k_1 + 9/40 \cdot k_2)), \\
k_4 &= f(t_n + 4/5h, y_n + h(44/45 \cdot k_1 - 56/15 \cdot k_2 + 32/9 \cdot k_3)), \\
k_5 &= f(t_n + 8/9h, y_n + h(19372/6561 \cdot k_1 - 25360/2187 \cdot k_2 + 64448/6561 \cdot k_3 \\
&\quad - 212/729 \cdot k_4)), \\
k_6 &= f(t_n + 1h, y_n + h(9017/3168 \cdot k_1 - 355/33 \cdot k_2 + 46732/5247 \cdot k_3 \\
&\quad - 49/176 \cdot k_4 - 5103/18656 \cdot k_5)), \\
k_7 &= f(t_n + 1h, y_n + h(35/384 \cdot k_1 + 500/1113 \cdot k_3 - 125/192 \cdot k_4 \\
&\quad - 2187/6784 \cdot k_5 + 11/84 \cdot k_6)), \\
b &= \{35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0\}
\end{aligned}
$$

The Dormand-Prince method is the currently used method in the simulation software to integrate the equations of motion for each molecule. Since it is a sensible default (used by default by mathematical software packages such as R and Matlab), and the problems with the simulation software do not arise from the numerical integration algorithm, we shall continue to use this method.

3.4 Comparison of ODE solving methods

To compare the effectiveness of different numerical methods of integration we shall look at convergence to the correct solution, and the computational cost of performing a single step. We express the computational cost to perform a step of a method which is integrating the function f as the number of evaluations of f that is necessary to perform this step. This is a valid way to measure the cost per step, since the time required to aggregate these results into the final step is often negligible compared to the time required for performing the function evaluations [6].

The simplest method that can be used to numerically solve an ODE is *Eulers Method*. If our ODE is of the form $\frac{dy}{dx} = f(x, y)$, then Eulers method states

that $y_{n+1} = y_n + h \cdot f(x, y_n)$ [6], for some small step size h. Eulers method exhibits first order convergence, but it requires only one function evaluation per step.

A method with greater order performance is the so called *Classical Runge-Kutta Method*. Here, the iterative method is given by the following steps [6].

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right), \\ k_1 &= f(x_n, y_n), \\ k_2 &= f(x_n + 0.5h, y_n + 0.5hk_1), \\ k_3 &= f(x_n + 0.5h, y_n + 0.5hk_2), \\ k_4 &= f(x_n + h, y_n + hk_3), \end{aligned}$$

Hence, it takes 4 function evaluations to perform a step of the classical Runge-Kutta method. Furthermore, this method exhibits fourth order convergence.

As we saw in Section 3.3, DORPI is a significantly more difficult method to perform. Performing one step of the DOPRI method requires 7 function evaluations, but in return the convergence is of fifth order.

In the simulation software, the step size is chosen in such a way that the largest error is the machine error using 64 bit precision floating point numbers. As laid out in the IEEE 754-2008 standard [7], these floats allocate 52 bits for their mantissa, yielding a machine epsilon of $2^{-53} \approx 10^{-16}$. In the simulation, the integration interval is chosen dynamically in such a way that the error of the integration is smaller than the machine epsilon. At such high accuracies, it is more important that the integration method converges quickly than that the number of function evaluations remains low. To illustrate this, consider Eulers method and the classical Runge-Kutta method. If we want to reduce the error of our integration by a factor of ten thousand, we need to divide h by $10,000$ for Eulers method. For the classical Runge-Kutta method however, we have fourth order convergence. This means that we need to divide h by only ten. Thus we see that for high accuracies the faster convergence outweighs the cheaper cost per step of the more slowly converging methods. Hence we conclude that since DOPRI has the highest rate of convergence, it is suitable for our needs.

3.5 Stark Effect

The molecules used in the experiment are neutrally charged, yet they can be influenced by an electric field. This is due to the occurrence of the Stark Effect [4]. Due to the fact that the orbiting electrons and the nucleus have an opposing charge, they experience an opposite force in an electric field. If this field is sufficiently strong, the molecule will obtain a dipole moment. In a diatomic molecule

where the atoms have a large difference in electronegativity, this means that the molecule will consist of two points with a different dipole moment. The force experienced by such a molecule is then given by $F = -\nabla E(\mathcal{E})$, as shown in [1].

4 Method

In this section we shall explain the methodology that is used to obtain results. We shall first explain the state of the software as it was, and then detail some modifications that were made.

4.1 Unmodified Setup

As the particles in the decelerator are simulated, the simulation code performs an acceptance check on the state of the particle. This means that it will perform three checks. First, it will check whether the particle has moved out too far in the radial direction from the center of the trap. This is to make sure that we discard particles that come too close to the rings, where the approximations to the electric field do not hold. Furthermore, we want to discard particles that collide with the rings. Secondly, it will check whether the particle has moved out too far to the front or back of the trap. Lastly it will check whether the particle has performed an adiabatic transition, and is thus no longer in the appropriate state.

For the purposes of this research, the simulation software was modified with an additional acceptance check. This check verifies whether the molecule has not gone below a supplied minimal threshold of the electric field. We perform this check because when the molecule approaches the zero of the electric field, the low field seeking state in which it was prepared is no longer guaranteed to be conserved. The reason for this is shown in Figure 2. In the case of a sufficiently low electric field, energy of the high field seeking state and the low field seeking state become close enough to allow the particle to transition between the states. Since our setup is created to trap particles in low field seeking states, the high field seeking states are unstable. This means that it is desirable for the particles to remain in electric fields that are high enough to make sure that the probability of transition between the states is low.

4.2 Waveforms

The waveforms in the simulation are restricted to harmonics of a pure sine wave. Several of these were tried. For each of these waveforms placed on the rings of the decelerator, we are interested in the shape of the the electric field.

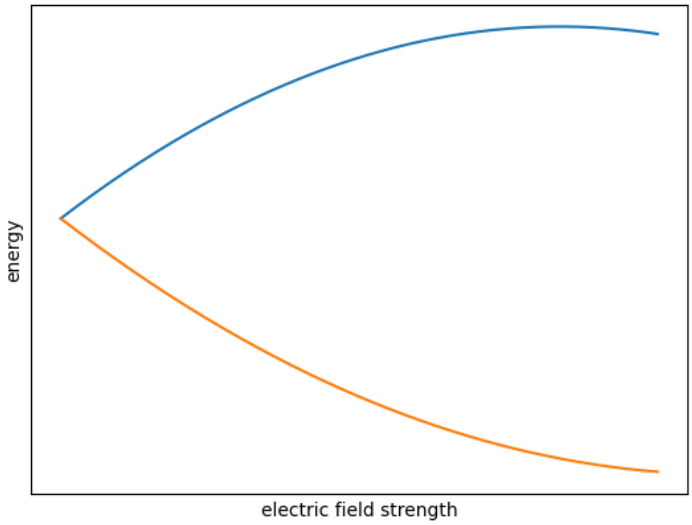

Figure 2: The splitting of spectral lines due to the Stark effect, showing two separate states. The blue line shows a low field seeking state, because if the electric field is lowered, this particle will decrease in potential energy. Conversely, the orange line shows a high field seeking state.

4.2.1 Pure Sine Wave

The simplest wave that we can try is a pure sine wave. The harmonics of a pure sine are simply given by

$$f(x) = \sum_{n=1}^{\infty} a_n \sin(nx) \text{ with } a_n = 1, 0, 0, ...$$

4.2.2 Triangle Wave

A slightly more complicated wave that we can use in the decelerator is a *Triangle Wave*.

$$f(x) = \sum_{n=1}^{\infty} a_n \sin(nx) \text{ with } a_n = (-1)^n (2n+1)^{-2}.$$

4.2.3 Sawtooth Wave

A sawtooth wave is a wave that rises linearly to a certain maximum, and then drops back down to its minimum. It is not realistic to construct a perfect sawtooth wave in the actual decelerator, because the discontinuity would require a very large voltage difference between the rings of the decelerator, which would cause a spark to form. Therefore, we only take the first 4 terms harmonics of the sawtooth wave. The harmonics are given by:

$$f(x) = \sum_{n=1}^{\infty} a_n \sin(nx) \text{ with } a_n = \frac{-1}{n}.$$

5 Results

In this section we shall show the results that have been obtained by running the experiments. The results are split into two categories. First we shall discuss the stability of the particles as they travel through the decelerator under various conditions. Then we shall discuss the results of tuning and rewriting parts of the code.

5.1 Stability

Simulations where run for different waveforms, with each simulation starting at $V_{initial}$ = 200 m/s, and varying V_{final}. We distinguish between two different modes for the decelerator. The first mode is called "guiding", which is where the decelerator is not actually used for deceleration, but instead the speed of the wave is kept constant. We expect this to result in the most stable configuration, so it will offer us a baseline to compare the stability against. Then, we will gradually decrease V_{final} to 0 m/s. The results of this are shown in Figure 3, for the sine

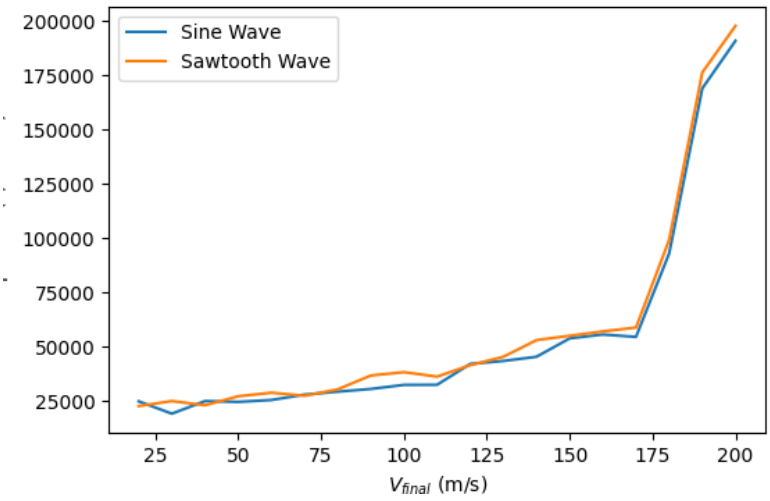

Figure 3: A graph showing the acceptance of multiple waveforms as a function of the desired deceleration speed. The acceptance is given in particles that reach the end of the detector per $5 \cdot 10^6$ particles.

wave and the sawtooth wave. The other waveforms are not included as they behaved more erratically and their acceptance was much lower.

Furthermore, in 3 it can also be seen that the sawtooth wave produces results that are marginally better than the sine wave. It was suspected that this was due to the fact that while the voltage on the rings may be sinusoidally shaped, the shape of the electric field does not necessarily need to be shaped in the same way. To investigate this, a simulation of the electric field in the decelerator was run for both the sine wave and the sawtooth wave. The code for these simulations is given in Appendix 7.1. The results of this are shown in Figures 4 and 5. Here it can be seen that the sine potential creates a neat set of traps, separated by a high electric field. The sawtooth wave however generates traps that are not always separated by "walls" of high electric field. Furthermore, when time evolution is taken into account, the traps seem to move through the "walls" that do exist, effectively dissipating them, and then emerging on the other side. It is therefore remarkable that the sawtooth even performs as well as it does.

One reason why we may get such a large acceptance above $V_{final} = 170$ m/s is that the incoming particles do not all have a $V_{initial}$ of precisely 200 m/s, but rather they form a distribution with 200 m/s as a mean. Since in the region 200 m/s $> V_{final} >$ 170 m/s we have that $V_{initial}$ is close to V_{final}, we may be

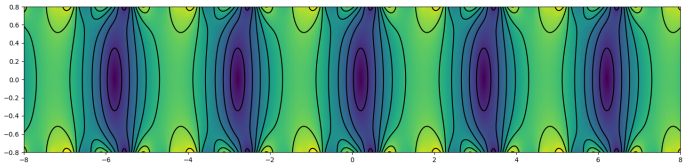

Figure 4: A schematic rendering of the electric field in the decelerator when the voltage on the rings is generated through a sine wave. Here a green colour indicates a strong electric field, whereas a blue colour indicates a weak electric field.

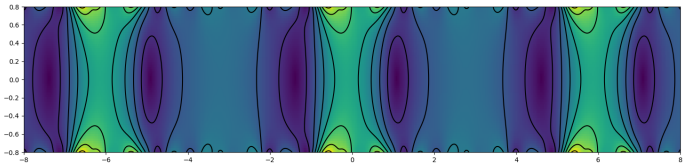

Figure 5: A schematic rendering of the electric field in the decelerator when the voltage on the rings is generated through a sawtooth wave. Again a green colour indicates a strong electric field, whereas a blue colour indicates a weak electric field.

measuring extra particles that already started out with the correct initial velocity. To investigate this, the sawtooth wave and sine wave were simulated again, this time with a $V_{initial}$ distribution shaped like a spike at 200 m/s. The results of this simulation are shown in Figure 6. Here we can clearly see that the acceptance of the sawtooth wave is much greater than the acceptance of the sine wave. Note also that the acceptance for both waveforms is much lower in the region of extreme deceleration. Furthermore we see that the acceptance of the sine wave when guiding is approximately the same as when the incoming particles follow a more realistic distribution, but the acceptance of the sawtooth wave has increased by a factor of 5.

As a final investigation, a simulation was run where the velocity of the incident particles was 350 m/s. From the simulation shown in Figure 6, we know that this velocity is too great to decelerate using a 4 metre decelerator. The results of this are shown in Figure 7. We note that the sine wave is able to decelerate particles up to a lower v_{final}, but the sawtooth is able to retain more particles than the sine for the same amount of deceleration.

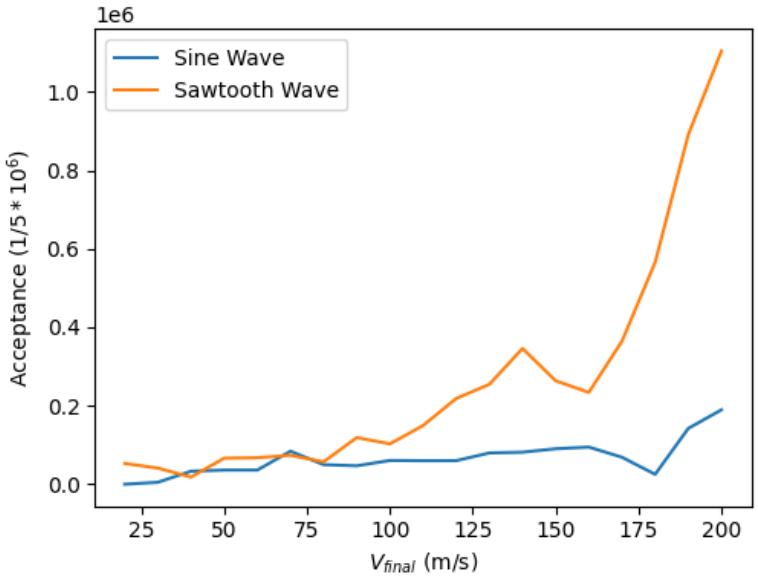

Figure 6: The acceptance of the Stark decelerator for the sine and sawtooth waveform. The vertical axis shows the number of particles that make it to the end of the decelerator when the incoming particles obey a velocity distribution that is a perfect spike around 200 m/s.

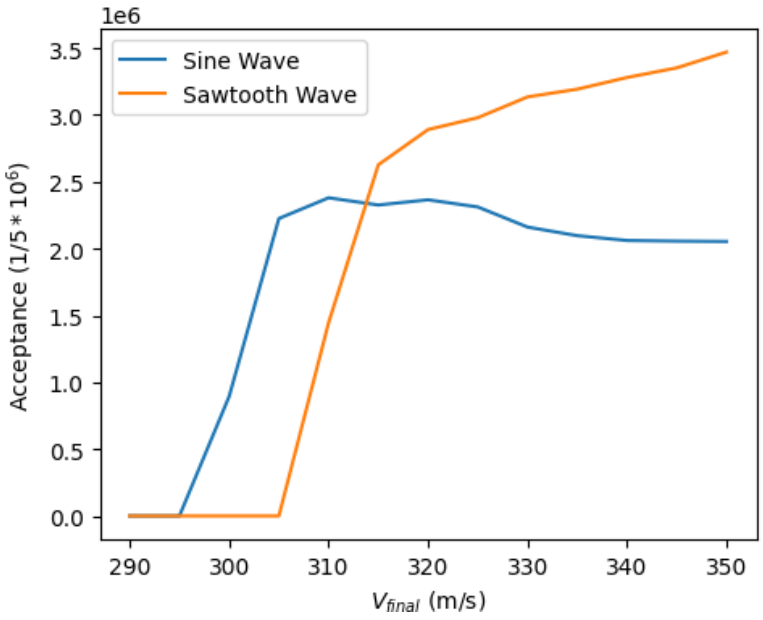

Figure 7: The deceleration profile of the sawtooth wave and the sine wave using $v_{initial} = 350$ ms. Note that the smallest v_{final} we can achieve is only 295 m/s, so the graph stops there.

5.2 Code

An attempt was made at improving the performance of the simulation code. The first method was to modify the C++ code in a way that could improve performance. A number of small issues were found in the C++ code, such the use of pass-by-reference of values smaller than the 64 bits, pass-by-copy of a few large objects and non-inlined functions. These were resolved, but a benchmark of the code showed that the difference in performance was not measurable.

The second method was to introduce greater parallelism. This is a nontrivial task because the implementation of Python contains a mechanism called the *Global Interpreter Lock* (GIL). This means that a single process of Python can only run a single line of code at once. There exist several projects that attempt to work around this limitation. The one used in the simulation software was called *Joblib*. In an attempt to run the simulations faster, the maximum number of parallel jobs was increased from 5 to 20. This introduced a segmentation fault approximately 1/3 times. Whether the segmentation fault occurred seemed independent of the simulation parameters, which indicates a *data race* occurring either in the C++ part of the simulations or in the internals of Joblib. To investigate, the simulation was run using Valgrind, which is a program that is able to detect where and how the memory leaks occur. Valgrind was not able to produce meaningful results about the cause of these segfaults.

After this, an attempt was made to rewrite the C++ code in a language called Rust, which is a recent language that is able to detect the presence of data races and invalid memory access during compilation of the program, refusing to compile any invalid programs. This is a significant advantage over using C++, especially considering that the performance characteristics of Rust and C++ are very similar. The drawback of using Rust instead of C++ is that the ecosystem for linear algebra and ODE libraries is much less mature at the time of writing. In the C++ a library called *Eigen* is used for vectors of fixed length, matrices and tensors. A library called *Nalgebra* was used in Rust for fixed vectors and matrices, but it lacks support for tensor contraction. Implementing a tensor product in a reasonably performant way is beyond the scope of this thesis.

6 Conclusion

We investigated the stability of the BaF molecules as they are decelerated using various waveforms. The runs using realistic simulation parameters did not show a significant difference between the currently used sine potential and the alternative potentials that were tried. We then reduced the noise produced in the simulations by introducing a more uniform input of particles. This then revealed a more significant difference between the sine potential and the sawtooth potential. After this, we measured the acceptance of the decelerator using particles that were too

fast to decelerate using a 4 metre decelerator. This showed that a sinusoidally shaped potential is able to decelerate particles to a lower velocity, and that a sawtooth potential is able to retain more particles when decelerating in a range that is accepted.

Since we ran into issues with the safety of the C++ code, an attempt was made at refactoring the C++ code into a safe language, namely Rust. The ecosystem of scientific computing using Rust proved to be not yet sufficiently mature to satisfy the requirements of the C++ code.

6.1 Further Research

There are a couple of areas where further research might yield interesting results. First, the sharp kink in Figure 3 is suspected to be due to the particles that are placed in the correct initial velocity by the BaF source, but this has not been verified. An attempt could be made at understanding this by creating a time evolution of the velocity of each particle. Secondly, implementing a tensor math library for Rust would allow the simulation code to take much greater advantage of parallelism, which would allow for faster simulations, or alternatively, simulations of a greater number of particles to produce more significant results.

7 Appendix

7.1 Electric field simulation

This code was created by Bart Schellenberg for his thesis, and slightly modified.

```
import math
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.animation as animation
import scipy.special as special
from numba import cuda

matplotlib.use('TkAgg')

# =================
# = Configuration =
# =================
nmax = 16
zbuff = np.linspace(-8, 8, 512)
sbuff = np.linspace(-.8, .8, 64)
d = .75
```

```
R = 1
aniFreq = .1
aniFrames = 600
aniFPS = 60

def sawtooth(phase, tau):
    return sum([1 / n * np.sin(.25 * np.pi * tau * n - phase) for n in range(1, 5)])

def funcAtau(phase):
    return [sawtooth(phase, tau) for tau in range(8)]

plt.plot([i for i in range(8)], funcAtau(0))
piover4d = np.pi / (4 * d)
size = (len(zbuff), len(sbuff))
aniSpeed = 2 * np.pi * aniFreq / aniFPS

# ====================
# = Prepare Buffers =
# ====================
# Bessel - can improve since i is symmetric
besselK0 = np.array([special.k0(piover4d * R * n) for n in range(1, nmax)])
besselI0 = np.array([[special.i0(piover4d * s * n) for s in sbuff] for n in range(1, nmax)])
besselI1 = np.array([[special.i1(piover4d * s * n) for s in sbuff] for n in range(1, nmax)])

# Result Buffer
fieldbuff = np.zeros(size)

# ==============
# = CUDA Stuff =
# ==============
bs = (16, 16)
bpg = (math.ceil(size[0] / bs[0]), math.ceil(size[1] / bs[1]))

zbuff = np.array(zbuff)
sbuff = np.array(sbuff)

@cuda.jit
def cuda_render(cZbuff,
                cSbuff,
                cFieldbuff,
                cAtau,
                cBesselK0,
                cBesselI0,
                cBesselI1,
                cPiover4d,
                cD):
    # Matrix index
    i,j = cuda.grid(2)
```

```
    size = cFieldbuff.shape

    # Skip threads outside the matrix
    if i >= size[0] or j >= size[1]:
        return

    # Render - ElectricStrength - (untag and tag "Render - Potential" to use)
    resultZ = 0
    resultS = 0
    for n in range(1, len(cBesselK0) + 1):
        interResZ = 0      # Intermediate result
        interResS = 0

        for tau in range(len(cAtau)):
            interResZ += cAtau[tau] * math.sin(cPiover4d * n * (cZbuff[i] - tau * cD))
            interResS += cAtau[tau] * math.cos(cPiover4d * n * (cZbuff[i] - tau * cD))
        resultZ += (interResZ * cBesselK0[n - 1] * cBesselI0[n - 1][j] * cPiover4d * n)
        resultS += (interResS * cBesselK0[n - 1] * cBesselI1[n - 1][j] * cPiover4d * n)
    result = math.sqrt(resultZ**2 + resultS**2)

    # Finish
    cFieldbuff[i][j] = result

funcRender0 = cuda_render[bpg, bs]
def funcRender(frame):
    funcRender0(zbuff, sbuff, fieldbuff,
                np.array(funcAtau(frame * aniSpeed)),
                besselK0, besselI0, besselI1, piover4d, d)

# =============
# = Animation =
# =============
fig, ax = plt.subplots(figsize=(18, 4))

def funcAnimate(frame):
    funcRender(frame)

    ax.clear()
    pcol = ax.pcolormesh(zbuff, sbuff, fieldbuff.T, linewidth=0, rasterized=True)
    pcol.set_edgecolor('face')

    ax.contour(zbuff, sbuff, fieldbuff.T, colors='black')

anim = animation.FuncAnimation(fig, funcAnimate, frames=aniFrames,
                               interval=1000.0/aniFPS, blit=False, repeat=False)
plt.show()
```

References

[1] (master thesis) Jeroen Muller - *Electric fields and molecule motion in a traveling-wave Stark decelerator*. University of Groningen, April 12th, 2017.

[2] (website) Rug EDM experiment - visited on April 27th, 2020 `https://www.rug.nl/research/vsi/newtopics/eedm`

[3] (publication) J.E. van den Berg, S.H. Turkesteen, E.B. Prinsen, and S. Hoekstra - *Deceleration and trapping of heavy diatomic molecules using a ring-decelerator*. June 2012

[4] (PhD thesis) Zapara, A. - *Dynamics of molecular beams in a traveling-wave Stark decelerator*. University of Groningen 2019

[5] (article) Sebastiaan Y.T. van de Meerakker, Hendrick L. Bethlem, Nicolas Vanhaecke, and Gerard Meijer - *Manipulation and Control of Molecular Beams*. Institute for Molecules and Materials, Radboud University Nijmegen, March 27th, 2012

[6] (book) Griffiths, David & Higham, Desmond. (2010). Numerical methods for ordinary differential equations. Initial value problems. 10.1007/978-0-85729-148-6.

[7] (website) `https://754r.ucbtest.org/background/`, visited on 16th July, 2020