To the Graduate Council:

I am submitting herewith a thesis written by Malachi Schram entitled "Implementation of Monte Carlo and Numerical Integration Techniques within an Online Physics Laboratory Environment". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Physics.

Marianne Breinig
Major Professor

We have read this thesis and
recommend its acceptance:

Michael W. Guidry
Wesley J. Hines

Acceptance for the Council:
Dr. Anne Mayhew
Vice Provost and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

Implementation of Monte Carlo and Numerical Integration Techniques within an Online Physics Laboratory Environment

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Malachi Schram

December 2002

Abstract

A robust and sophisticated online physics laboratory environment has been developed. This environment can handle large data sets and generate realistic experimental results by applying Monte Carlo and numerical integration techniques. The advantages and limitations of both the Flash 5 and Java development environments were explored. Java was chosen for it's ability to handle large data sets and consequently used to create the Java Laboratory (JLab) environment. Within the online environment two JLabs were created, the "Online Virtual Nuclear Decay Laboratory" and the "Online Virtual Stern-Gerlach Laboratory". These laboratories teach students how to manipulate experimental parameters, take data, and use various analysis tools. These JLabs generate realistic data sets for students to analyze and prove that online laboratories can play a significant role in enhancing physics education.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The primary goal of this thesis is to develop a robust and sophisticated online physics laboratory environment. For the purpose of this thesis, we define a robust online environment as one that can handle large data sets. In addition, we define a sophisticated online environment as one in which Monte Carlo and numerical integration techniques are applied to create realistic experimental results. In this thesis, the following Monte Carlo techniques are used to create the online physics laboratory environment:

- Inverse Transformation Method
- Rejection Method
- Smearing Method

In addition, Euler's numerical integration technique is applied within the online physics laboratory environment.

This thesis also compares the Flash 5 and Java development environments to

determine which is best suited to develop the online physics laboratory environment.

1.1 Online Physics Learning

Most students taking introductory physics classes at universities, such as the University of Tennessee, are not physics majors. Physics departments offer a diverse number of courses covering basic physical principles and applications. Students attending these courses come from a broad range of disciplines such as pre-medical, engineering, and art and sciences. The physics classes at research universities are frequently very large, with two or more parallel sessions. Much of the learning takes place through formal lectures, which educational research has repeatedly shown to be the least effective way to impart knowledge [1]. Laboratory exercises are often of the "cookbook" type and are not well integrated with the lectures.

Many of the new approaches to improve student learning in introductory physics education require the integration of lectures and laboratories [2, 3]. However, large class sizes and constraints on laboratory space and equipment often do not allow this integration. Online virtual laboratories can be integrated with lectures to help students learn science by providing a direct experience with the methods and processes of inquiry. These virtual laboratories can be made available to students anywhere, at any time, through the World Wide Web. Students will be able to complete their laboratory assignments without the use of special equipment or software, simply using a freely available web browser. Evidence already exists showing that modifying physics laboratories by adding online conceptual questions has increased the average student's learning [4]. It has also been suggested that creating Internet-based "virtual physics" experiments would be most

beneficial when addressing safety concerns or limited equipment availability [5]. We expect that the availability of such experiments will increase student learning and help reach a broader target audience.

Many educational web sites have been created in recent years and much effort has gone into enhancing the online learning environment. Presently, most online physics learning environments teach students how to understand theoretical models by using visualization techniques [6]. The goal of this thesis is to add an experimental component to the environment and thereby enhance the online learning experience. The online laboratories developed in this thesis will allow students to perform experiments and obtain realistic data via the use of robust and sophisticated simulation techniques. Students will learn how to setup experimental parameters and analyze the resulting data.

The University of Tennessee has funded the Center for Advanced Educational Technology (CAET) to address the need to develop Internet resources for education. CAET has three primary themes [7]:

- Develop and implement second[1] and third[2] generation educational technologies
- Train graduate and undergraduate students in emerging educational and informational technologies
- Involve graduate and undergraduate students with faculty members to im-

[1]Interactive modules that respond to user input, require extensive programming, and are accessible from the Web.

[2]Interactive modules that show promise of intelligent adaptation to student's input and are accessible from the Web.

plement high-profile advanced projects using those technologies

CAET is an ideal working environment to achieve the goals of this thesis. The development of an online laboratory will address all of the CAET themes.

- *First Theme*: We are developing workable second-generation educational technologies.
- *Second Theme*: We are training a graduate student, Malachi Schram.
- *Third Theme*: We are involving the graduate student Malachi Schram with a faculty member, Dr. Marianne Breinig.

1.2 Online Physics Laboratories

Listed below are two online physics laboratories that have been developed for this thesis:

- Nuclear Decay Experiment
- Stern-Gerlach Experiment

The nuclear decay laboratory is a standard introductory university physics laboratory, which can be quickly integrated into the curriculum. Nuclear decay is inherently probabilistic and the outcome of the decay experiment is ideally suited to incorporate Monte Carlo techniques. The online nuclear decay laboratory demonstrates the use of the inverse transformation method, the rejection method, and the smearing method.

The Stern-Gerlach experiment must be developed using both classical and quantum physics. In addition to demonstrating the use of Monte Carlo techniques in

the nuclear decay laboratory, we show how to apply Euler's numerical integration technique to solve the classical equations of motion. By developing this laboratory, we show that upper-level undergraduate experiments can be developed for online use.

This thesis is organized in the following manner: Chapter 2 introduces the modeling techniques used within this thesis. Chapter 3 compares the advantages and limitations of the Flash 5 and Java development environments for producing online laboratories. The development of the individual JLabs is explained in chapters 4 and 5. A summary of the results and suggestions to further enhance the Java laboratories can be found in chapter 6.

Chapter 2

Physics Modeling Techniques

2.1 Monte Carlo Techniques

Monte Carlo techniques are statistical methods used to solve physical or mathematical problems. These techniques use sequences of random numbers to perform simulations. Monte Carlo techniques are ideally suited to simulate random or stochastic processes, since these processes can be described by probability density functions. Therefore, physics processes that can be described using a probability density function can be simulated directly without writing down the differential equations that describe their behavior.

Monte Carlo techniques can also be applied to solve problems that appear to have no stochastic content, such as evaluating a definite integral. Therefore, Monte Carlo techniques can be used as long as one can describe the problem in terms of a probability density function.

In this thesis, we achieved realistic simulations for the online physics laborato-

ries by applying various Monte Carlo techniques. The following sections describe these techniques.

2.1.1 Inverse Transformation Method

The inverse transformation method [8, 9] is used to randomly generate a probability density function by solving the probability density function's variable in terms of randomly generated numbers. This method is the most commonly used method for simulating a random variable with an exponential probability density function. To understand the inverse transformation method, we need to represent the random number generator in terms of a probability density function. Most computer programming languages have development packages[1] that provide a random number generator for a uniform distribution of pseudo-random numbers between 0 and 1. Hence, we introduce the following function as our uniform probability density function:

$$\rho(x) = \begin{cases} 1, & 0 \leqslant x \leqslant 1 \\ 0, & \text{elsewhere} \end{cases} \tag{2.1}$$

The above probability density function is normalized as illustrated below:

$$\int_{-\infty}^{\infty} \rho(x)\, dx = 1 \tag{2.2}$$

From the uniform probability density function, we can express the random number as follows:

$$r = \int_{-\infty}^{r} \rho(x)\, dx \tag{2.3}$$

To illustrate how the inverse transformation method is applied, we need to turn our attention to an arbitrary probability density function $f(s)$. If we integrate

[1] Java offers a method called Math.random() [10].

$f(s)$ up to an arbitrary point a, we get:

$$F(a) = \int_{-\infty}^{a} f(s)\, ds \tag{2.4}$$

To ensure conservation of probability, we set:

$$\int_{-\infty}^{r} \rho(x)\, dx = \int_{-\infty}^{a} f(s)\, ds \tag{2.5}$$

Therefore, we can express $F(a)$ in terms of r.

$$r = F(a) \tag{2.6}$$

Provided that we can solve the inverse of $F(a)$, we can generate a unique variable a as follows:

$$a = F^{-1}(r) \tag{2.7}$$

The following example explains how to apply the inverse transformation method to an exponential function.

$$f(s) = \begin{cases} \frac{e^{-\frac{s}{\tau}}}{\tau}, & 0 \leqslant s < \infty \\ 0, & \text{elsewhere} \end{cases} \tag{2.8}$$

Integrating the function $f(s)$, we get:

$$F(t) = \int_{-\infty}^{t} f(s)\, ds = e^{-\frac{t}{\tau}} \tag{2.9}$$

For our example, the solution to 2.6 is:

$$r = e^{-\frac{t}{\tau}} \tag{2.10}$$

The final step requires us to solve 2.10 in terms of the random number r. In our example, this yields the randomly generated variable t, given below.

$$t = -\tau ln(r) \tag{2.11}$$

The inverse transformation method can always be used to generate random numbers with a distribution F, provided one can calculate the inverse F^{-1}. The method is therefore limited to probability density functions that can be integrated analytically, unless we wish to use numerical techniques to solve the integral. However, other Monte Carlo techniques are then better suited to generate the desired distribution. Some of these techniques will be discussed in the following sections.

2.1.2 Rejection Method

The rejection method [9, 11] is a technique used to generate random numbers for a known and computable distribution function $f(x)$. To generate the distribution function $f(x)$, we need to define a simple finite distribution function $h(x)$, such that it encloses the desired distribution function $f(x)$, as illustrated in figure 2.1. We should note that this comparison function $h(x)$ can be any arbitrary finite function, as long as it lies above the curve of $f(x)$. Once $h(x)$ has been defined, we have to choose uniformly distributed random points under $h(x)$. Whenever a point lies outside the area under the probability density function $f(x)$ we reject it and choose another point.

To optimize the simulation speed, one should carefully choose the finite distribution function $h(x)$. The following should be simultaneously considered when defining $h(x)$.

- *The* $\frac{f(x)}{h(x)}$ *area ratio*: Comparing the area of both functions indicates the fraction of points that will be accepted.

- $h(x)$ *generating time*: Although $h(x)$ can be any arbitrary finite function, it is wise to select a function that is simple enough to quickly generate results.

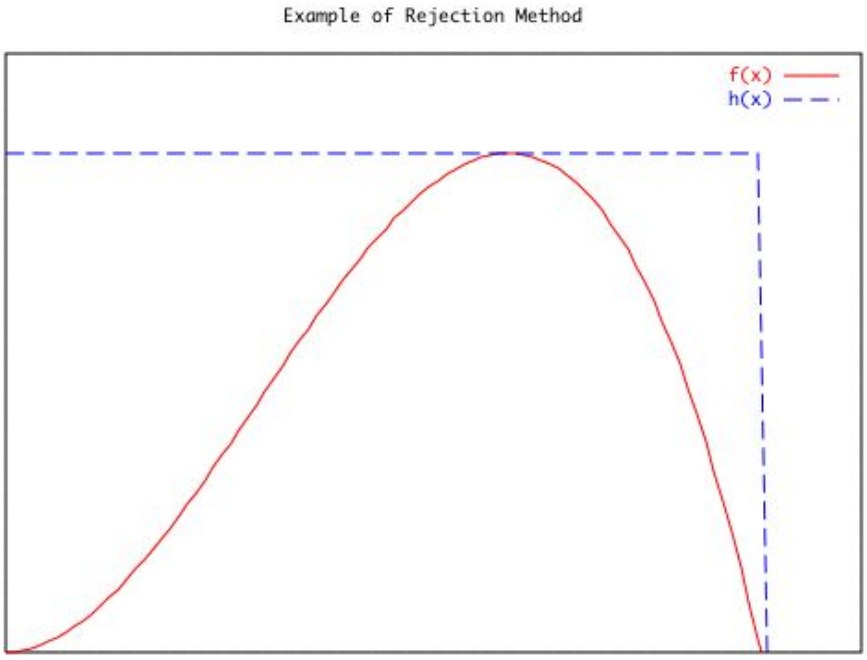

Figure 2.1: Illustration of a possible distribution function $f(x)$ and comparison function $h(x)$ for the rejection method.

To choose a random point for x, we can use a variant of the inverse transformation method described in section 2.1.1, provided that the function $h(x)$ is analytically known and invertible. Let A denote the total area under $h(x)$. We generate a random number r, to be used to produce a number between 0 and A. We let:

$$rA = \int_0^{x_i} h(x)dx \tag{2.12}$$

and solve for x_i. Now we generate a second random number between 0 and $h(x_i)$. If $rh(x_i) \leqslant f(x_i)$ we accept x_i, if not we reject x_i.

Let us use the Maxwell speed distribution as our example[2]:

$$\rho(v) = \begin{cases} 4\pi \left(\frac{m}{2\pi kT}\right)^{3/2} v^2 e^{-mv^2/2kT}, & 0 \leqslant v < \infty \\ 0, & \text{elsewhere} \end{cases} \tag{2.13}$$

Here $\rho(v)$ represents the distribution of speed for molecules in an ideal gas, where m is the molecular mass, k is the Boltzmann constant, T is the temperature , and v is the speed of the molecule. In order to determine the maximum value of the Maxwell speed distribution, we need to determine the most probable speed:

$$v_{mp} = \frac{d\rho}{dv} = \sqrt{\frac{2kT}{m}} \tag{2.14}$$

Next, we need to determine the boundary condition that we will impose on the variable v. The boundary condition stated in our function can not be used given that we would be running the simulation for an infinite amount of time. Therefore, we need to apply an acceptable boundary condition such that the simulation time and the validity of the data are optimal. We use v_{mp} as a reference point and apply a scale factor "ξ" such that we have our boundary set at $v_{max} = \xi v_{mp}$. For our comparison function $h(v)$, we use:

$$h(v) = \begin{cases} \rho(v_{mp}), & v \leqslant v_{max} \\ 0, & \text{elsewhere} \end{cases} \tag{2.15}$$

This is the simplest possible comparison function that encloses the distribution $\rho(v)$. Next, we generate a random speed v_r, between 0 and v_{max}, and calculate $\rho(v_r)$. Finally, we generate another random number r between 0 and 1 and determine if $rh(v_r) \leqslant \rho(v_r)$. If this is true, v_r is a valid speed.

[2]We should note that we do not have to worry about normalizing the function.

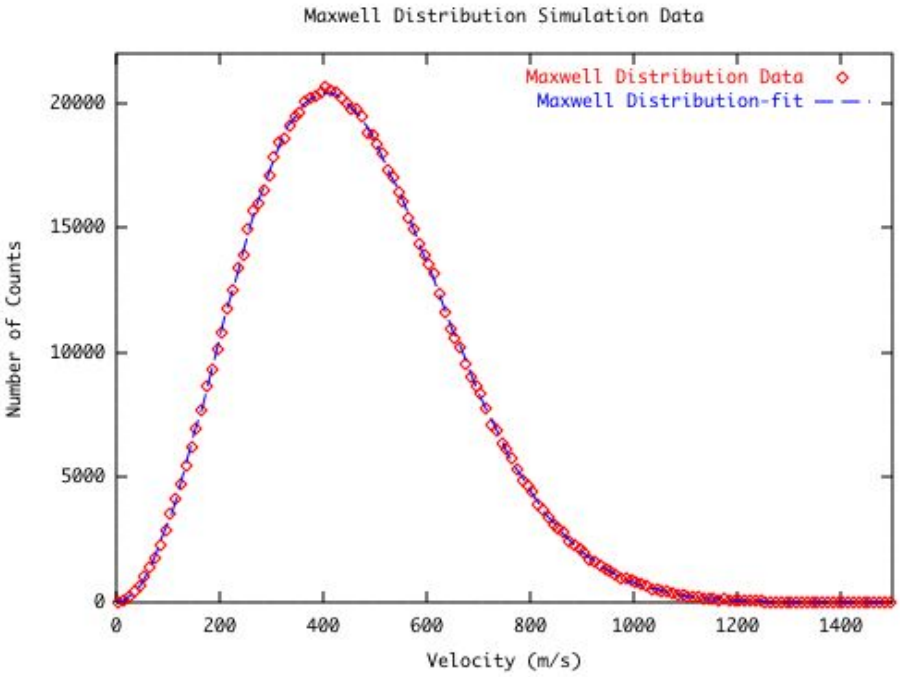

Figure 2.2: Speed distribution of potassium atoms at 388.15K, generated using the rejection method.

To validate that we have properly applied the rejection method, we need to analyze the simulated results with the distribution given in equation 2.13. In figure 2.2, we compare one million randomly generated events with the theoretical distribution. The fitting procedure requires redefining equation 2.13 such that $\alpha = \gamma 4\pi \left(\frac{m}{2\pi kT}\right)^{3/2}$ and $\beta = m/2kT$, where γ is a scaling factor that is dependent on the number of events generated. Therefore, we get:

$$\rho(v) = \begin{cases} \alpha v^2 e^{-\beta v^2}, & 0 \leqslant v < \infty \\ 0, & \text{elsewhere} \end{cases} \tag{2.16}$$

Table 2.1: Comparison between simulation data and the Maxwell speed distribution.

Variable	Theoretical Value	Fitted Value	% Error
β	6.057E-6	6.061E-6±5.371E-9	0.1491

Next, we fit[3] the simulated data to equation 2.16 with α and β as adjustable parameters. The results of the fit can be found in table 2.1. We see that the simulated data closely represents the Maxwell speed distribution. The advantage of the rejection method is that we do not have to solve any equations. However, it is inefficient given that we have to generate many events that are rejected. Due to this disadvantage, applying good boundary conditions is crucial.

2.1.3 Smearing Method

In this thesis, the smearing method [8] is used to generate deterministic experimental results, including experimental uncertainties. In order to use this method, we must first assume that the data we are modeling have a Gaussian distribution.

$$\rho(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.17}$$

Here μ is the mean and σ is the standard deviation. Next, we must calculate the theoretical value of the deterministic experimental results and apply an uncertainty by smearing the data.

$$x = \mu + \sigma g \tag{2.18}$$

Here x is the smeared result, μ is the calculated theoretical value , σ is the standard deviation, and g is the random sample from the Gaussian distribution. The

[3]All fits in this thesis were conducted using the GNU Plot version 3.7.1d software, which applies the nonlinear least-squares Marquardt-Levenberg algorithm.

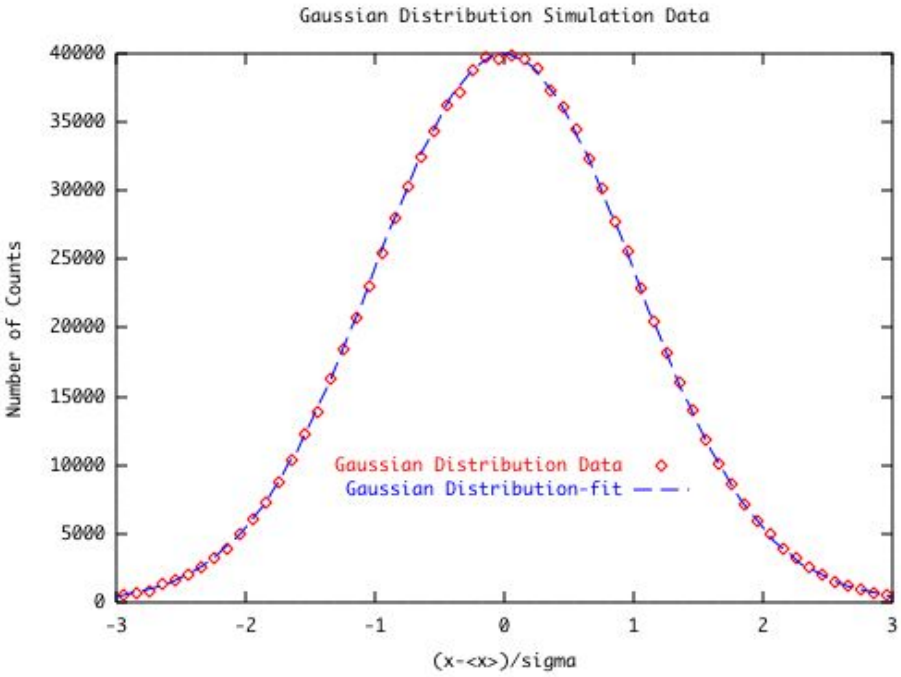

Figure 2.3: Gaussian distribution with $\mu = 0$ and $\sigma = 1$, generated using the rejection method.

rejection method discussed in section 2.1.2 is used to generate the random sample.

Figure 2.3 represents the results of one million randomly generated events from the Gaussian distribution. The source code is located in appendix B.1. Applying the same analysis technique as in section 2.1.2, we conclude that the simulation closely represents the Gaussian distribution. Results of the fit can be found in table 2.2.

Let us use an example to illustrate the smearing method. Suppose we are try-

Table 2.2: Comparison between simulation data and the Gaussian distribution.

Variable	Theoretical Value	Fitted Value	% Error
σ	1	1.00112 ± 9.4E-4	0.2058

ing to measure the electric force between two point charges using the following theoretical equation.

$$F(x) = -\frac{kq_1q_2}{x^2} \tag{2.19}$$

However, as we measure along the $\hat{x}$-axis, our measurements will have an uncertainty due to the finite resolution of our instrument. Let us assume that the uncertainty is defined such that $\sigma = 0.25$. When we simulate this function, we can simply add a smearing effect to the position measurement as follows:

$$x_i' = x_i + \sigma_i g_i \tag{2.20}$$

We then take the smeared position measurement and insert it into equation 2.19. The results are listed in table 2.3 and plotted in figure 2.4.

Table 2.3: Results of the electric force simulation, where $q_1 = q_2 =$ fundamental charge constant.

x_i [m]	x_i' [m]	$F_i(x_i')$ [N]
1	1.2136	$1.5629E-28$
2	2.2493	$4.5494E-29$
3	2.5931	$3.4232E-29$
4	3.8587	$1.5459E-29$

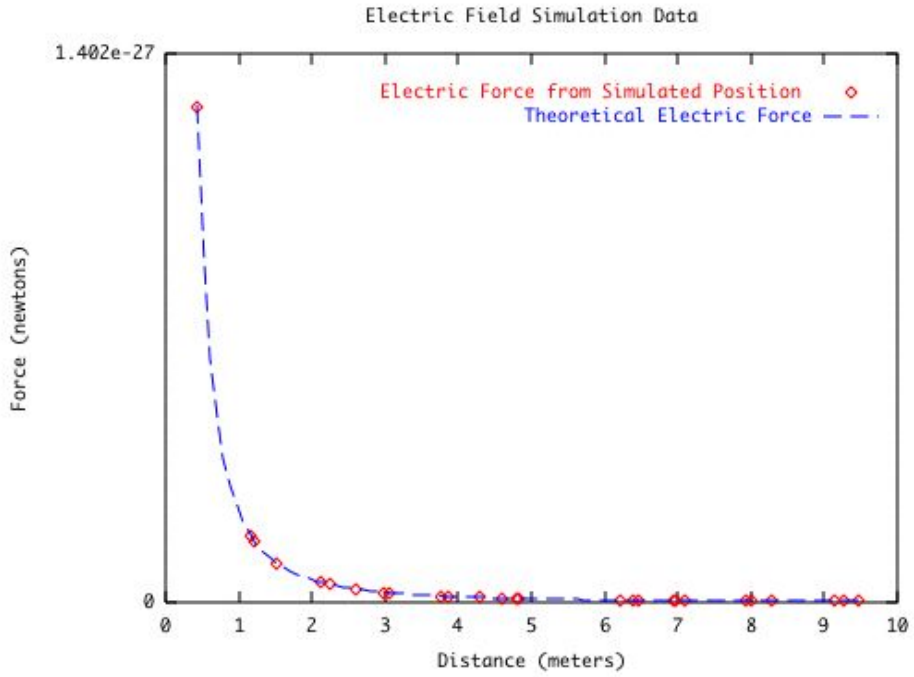

Figure 2.4: Results of the electric force simulation, generated using the smearing method.

2.2 Euler's Method, a Numerical Integration Technique

The computational simulations that we are running require that we solve the necessary equations of motion. Euler's method [12] is an approximation technique commonly used to solve the equations of motion [13]. Let us look at Newton's Second Law of Motion:

$$\vec{F} = m\frac{d\vec{v}}{dt} \tag{2.21}$$

$$\frac{d\vec{v}}{dt} = \frac{\vec{F}}{m} \tag{2.22}$$

$$d\vec{v} = \frac{\vec{F}}{m}dt \tag{2.23}$$

Now, imagine that we have a particle moving in the $\hat{x}$-direction through a gravitational field in the $\hat{z}$-direction. We can define the equations of motion in the $\hat{z}$-direction as follows:

$$dv_z = -gdt \tag{2.24}$$

Here g is the Earth's gravity constant. When applying a numerical integration to equation 2.24, we are approximating an infinitely small increment of time as follows, $dt \rightarrow \Delta t$ and $dv \rightarrow \Delta v$:

$$\Delta v = -g\Delta t \tag{2.25}$$

Hence, to determine the velocity of the particle after a small time increment, with the initial condition $v(t)$, we would get:

$$v(t + \Delta t) = v(t) - g\Delta t \tag{2.26}$$

In order to understand the level of accuracy of Euler's method, we must apply Taylor's theorem [14] to the function $v(t + \Delta t)$:

$$v(t + \Delta t) = v(t) + (\Delta t)\frac{dv}{dt} + \frac{(\Delta t)^2}{2!}\frac{d^2v}{dt^2} + \cdots \tag{2.27}$$

If we compare equation 2.26 with equation 2.27, we notice that Euler's method goes out to the first order term and truncates all higher terms. Therefore, Euler's method provides a first order approximation with a second order truncation error.

Chapter 3

Technical Environments and the Java Laboratory

3.1 Technical Environments

In this thesis, we consider two possible development environments in which to create an online laboratory [15, 16].

- Macromedia Flash 5
- Sun Microsystems Java

The following is a brief description of these environments.

Macromedia Flash 5

Macromedia Flash 5 is a user friendly software package designed to create web based animations and programs. The Flash 5 environment can be divided into two primary components, the graphical development environment and the scripting language ActionScript [17].

The graphical development environment is used to draw pictures, create animated movies, and, in our case, create a graphical user interface. To add functionality, Flash 5 offers the scripting language ActionScript, similar to that of JavaScript [18]. ActionScript provides programmers with the ability to create interactive web applications by associating scripts to pictures or movies on the Flash 5 web page. We can look at the ActionScript script as an executable program that is associated with a picture created inside the Flash 5 environment. For example, if we have a picture that represents a button, the script is executed when the user clicks on the button. ActionScript provides more control over Flash 5 objects than the graphical development environment alone, as well as the ability to write programs associated with a Flash 5 movie [18].

Sun Microsystems Java

Java is an object-oriented interpreted language used to write applications and applets. To create a Java application or applet, one must first write the program code. The program code is then compiled into an intermediate portable format, which can be interpreted by a platform dependent software. Slower performance is inherent with interpretive languages. However, this drawback is greatly outweighed by having a cross-platform environment. The beauty of Java is that it is truly cross-platform, as pointed out in the slogan "Write Once, Run Anywhere" [10]. This is a key factor when considering the development of academic online software. Academic software should be available to everyone, regardless of the operating system. Furthermore, the newer releases of Java[1] perform almost as fast as programming languages such as C and C++ [10]. Hence, the Java envi-

[1] Java 1.2 or higher.

ronment should not limit the development of numerically intensive simulations[2].

3.2 Environment Comparison

To achieve the goals of this thesis, we devoted some time to understanding the advantages and limitations of both the Flash 5 and Java environments. We wanted to make sure that these environments satisfy the following objectives which we consider crucial for the development of online laboratories:

- cross-platform
- generation of true simulation models
- expandable

The following paragraphs discuss the capabilities of both environments, taking into consideration the objectives stated above.

Flash 5 was selected as the first environment to be investigated, given its obvious advantages[3]:

- short learning curve
- ease in creating graphical objects
- minimal programming requirements

The first laboratory developed using Flash 5 was the nuclear decay laboratory[4]. Introductions to the theory of nuclear decay and to the experimental methods

[2]Although numerically intensive simulations may not be a problem for the Java environment, we need to consider the limitation of the end-user's computer.

[3]These advantages equate to a quicker development time.

[4]A detailed explanation of the experimental goals can be found in chapter 4.

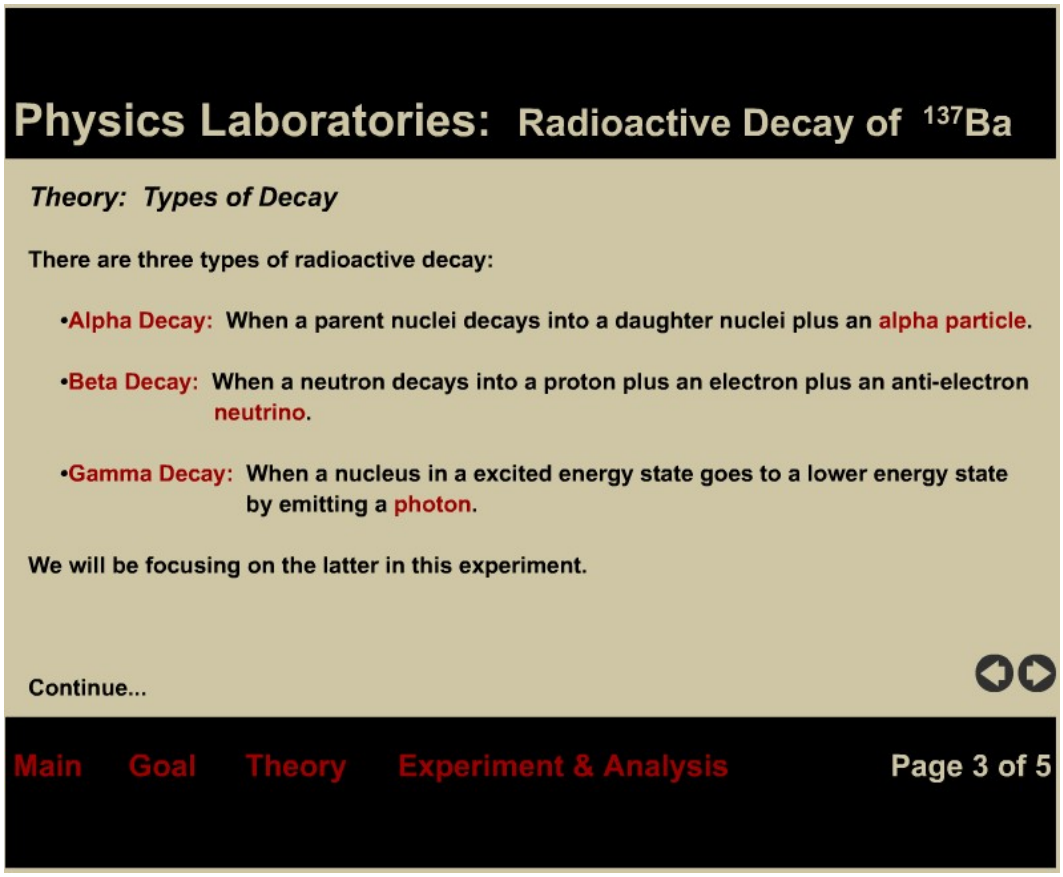

Figure 3.1: Explanation of the nuclear decay theory displayed using Flash 5.

used in measuring nuclear decay were developed using Flash 5. These introductions were written as a series of pages. Every page was designed to focus on one core physics concept. Figure 3.1 is an example of such a page. Some parts of the text were highlighted in red to indicate that further explanations were available. These explanations where delivered in the form of Flash 5 animations. An example of a Flash 5 animation is shown in figure 3.2. The student must progress through the pages in an orderly fashion and study all the core concepts before he/she can perform the experiment.

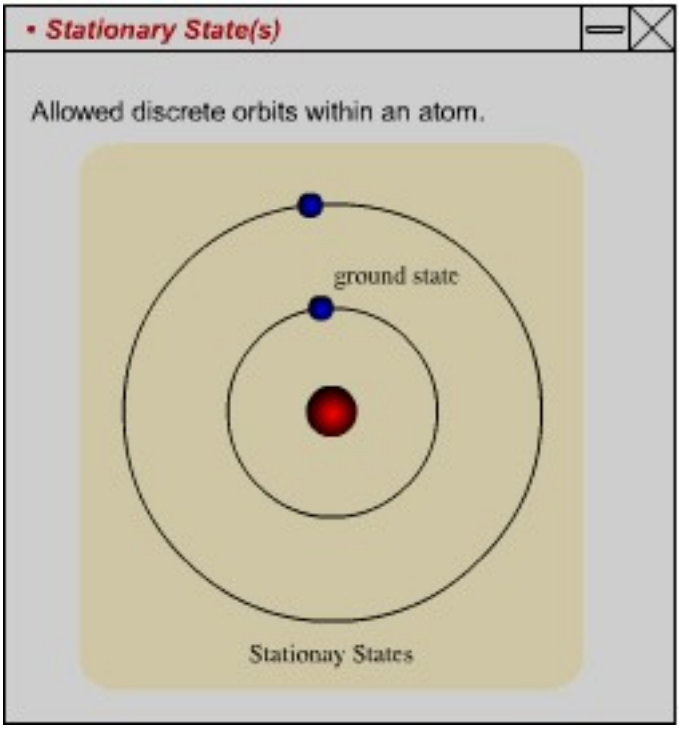

Figure 3.2: Explanation of a stationary state using a Flash 5 animation.

Figure 3.3 shows a snapshot of the Flash 5 nuclear decay laboratory GUI. While Flash 5 was well suited to create the GUI, the limitations of ActionScript became apparent. When generating large numbers of random nuclear decays, we experienced a reduction in simulation speed[5]. This presents a serious limitation in the development of Monte Carlo or numerical integration techniques used for online laboratories.

Next, we developed a similar nuclear decay laboratory using Java[6]. In comparison to Flash 5, Java has a much longer learning curve. It has no graphical development environment. Therefore, the graphical user interface has to be programmed.

[5]See section 4.2.1 for an explanation on generating random nuclear decay events.

[6]See chapter 4 for a complete explanation of the laboratory.

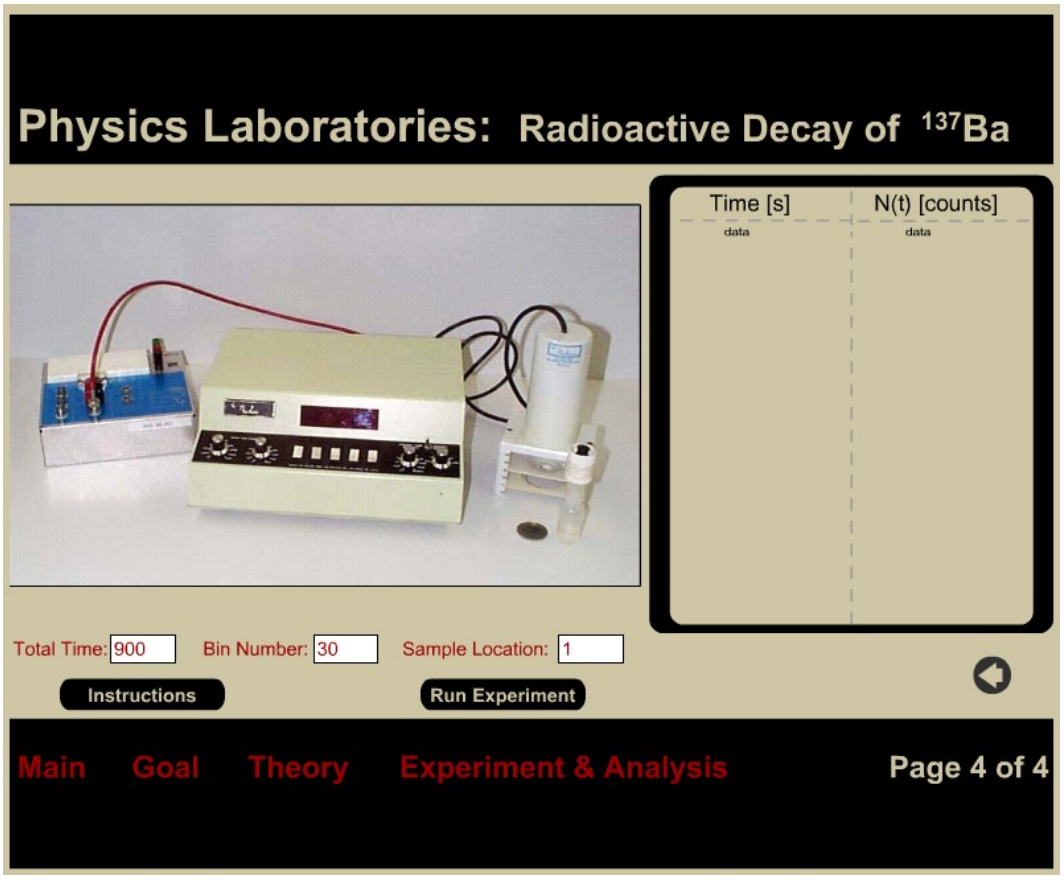

Figure 3.3: Nuclear decay experiment GUI displayed using Flash 5.

However, Java has no problem generating a large number of random events.

Given that generating realistic experimental results using Monte Carlo techniques is one of the main goals of this thesis, we decided to use Java to further develop the online laboratories. The capabilities of Flash 5 to produce realistic and timely simulations were too limited.

3.3 Java Laboratory

To achieve the goal of this thesis, we developed a Java Laboratory (JLab). JLab is designed to be an expandable online software package used to simulate physics experiments. All physics processes and supporting code were developed using the object-oriented paradigm. The following topics were considered when developing the JLab environment.

- *Ease in developing new online experiments*: Similar to C++, Java has inheritance, therefore developers can enhance pre-existing physics objects to create new experiments without modifying the original source code.

- *Ease in adding new functionality to existing JLabs*: Modifications to pre-existing experiments can be accomplished by calling new objects to implement new functionality.

We used the Java predefined class JApplet[7] to develop the graphical user interface (GUI), which allows the end user to manipulate the various components of the online laboratory.

3.3.1 Graphical User Interface Structure

The GUI for the JLab is divided into two primary components: the laboratory GUI option panel and the HTMLViewer. Both components are JPanels[8], where the first component gives the user access to the variables of the laboratory and the second component is used to present html files.

[7]The JApplet class is used to develop software applications intended for the Internet.

[8]Predefined Java class used to organize the location and behavior of each individual GUI component.

Laboratory GUI Option Panel

The first component is a laboratory specific GUI option panel used to provide the user with the ability to manipulate the experiment parameters, such as the temperature in an oven or the strength of an applied magnetic field. We will further discuss this topic in sections 4.3 and 5.3, given that each laboratory has a specific laboratory GUI option panel.

HTML Viewer

To present the theory and the experimental procedures to the student through the JLab environment, we have developed a Java object called the HTMLViewer. The HTMLViewer object was created to present this information in an html format by handling the connection and processing of the html files. HTML pages can be generated using various software packages or even a text editor. Furthermore, we developed this object with the intent to use it for further JLab experiments.

The HTMLViewer.class has one constructor[9] and three methods[10]. The HTMLViewer() constructor takes two String arguments. The first is the title of the html viewer and the second is the url of the html file. Once the HTMLViewer is initialized and the two String variables are defined, it calls on the getHTMLViewer() method. A sample of the HTMLViewer() constructor is given below:

```
public HTMLViewer(String getTitle, String getURL) {
        this.stringURL = getURL;
        this.HTMLtitle = getTitle;
        getHTMLViewer();
}
```

[9] A constructor is a segment of Java code that is called to define the class object.

[10] A method is a segment of Java code that tells the program to perform a specific task.

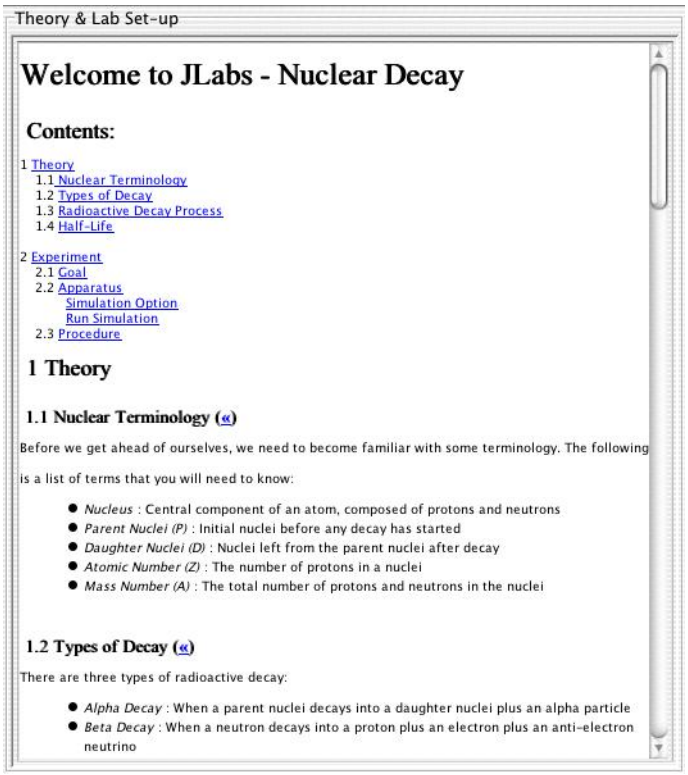

Figure 3.4: Explanation of the nuclear decay theory displayed using the HTMLViewer class.

The getHTMLViewer() method was developed to setup the appearance of the html viewer by defining the title and the border layouts. It then calls on the createHTMLPanel() method which verifies the location of the html and publishes the valid html. The third method is createHyperLinkListener(), used to handle any links within the html. Figure 3.4 shows a GUI built using the Java code HTMLViewer.class. The source code is available in appendix C.1. Java has a package[11] called *java.net* that handles the URL connection [10]. In addition, the package *java.text.html* allows us to process the html document [19].

[11] A Java package is a set of related predefined classes.

3.3.2 How does a Java Laboratory Work?

In order to complete an online laboratory, we used Java's predefined JApplet class to present the laboratory on the Internet. This is achieved by calling both the HTMLViewer and the laboratory GUI classes. See figure 4.5 in section 4.4 for an example of a JLab. The following is a chronological example of how the JLab should work:

- A student logs on to the desired JLab.
- The JApplet is initialized, along with all necessary classes.
- The student reads the theory and experimental procedures presented by the HTMLViewer.
- The student manipulates the input parameters of the laboratory GUI according to the instructions.
- The laboratory GUI relays the information to the JApplet.
- The JApplet redefines the variables that were manipulated by the student.
- Once the student is satisfied with the input parameters, he/she clicks on the "run" button.
- The JApplet calls the necessary methods to generate the simulation data.
- Once the simulated data is generated, the JApplet returns an output to the student.
- The student can analyze the data by using the built in analysis tools or by importing the results into their preferred analysis software.

Unlike the Flash 5 version of the nuclear decay laboratory, the Java version displays the theory and experimental procedures on a single html page. This html page is designed with quick links to access specific topics, without leaving the web page. Therefore, students will have immediate access to the theory and experimental procedures, without having to go back and forth as with the Flash 5 version.

The following chapters will present two JLab case studies.

Chapter 4

Java Laboratory 1: Nuclear Decay

As stated in chapter 1, the goal of this thesis is to develop a robust and sophisticated physics laboratory simulation within an online environment. The focus of the nuclear decay JLab is to create an interactive laboratory using various Monte Carlo techniques.

This JLab represents a variation of a typical second year undergraduate nuclear decay laboratory. An illustration of the hands-on nuclear decay laboratory is provided in figure 4.1. The goal of a traditional hands-on nuclear decay laboratory involves studying a radioactive source and determining its half-life [20]. This is accomplished by detecting γ-radiation emitted by the nuclear decay using a Geiger-Müller tube. The following are traditional objectives for this experiment:

- To investigate the radioactive decay
- To understand the concepts of nuclear activity and nuclear half-life

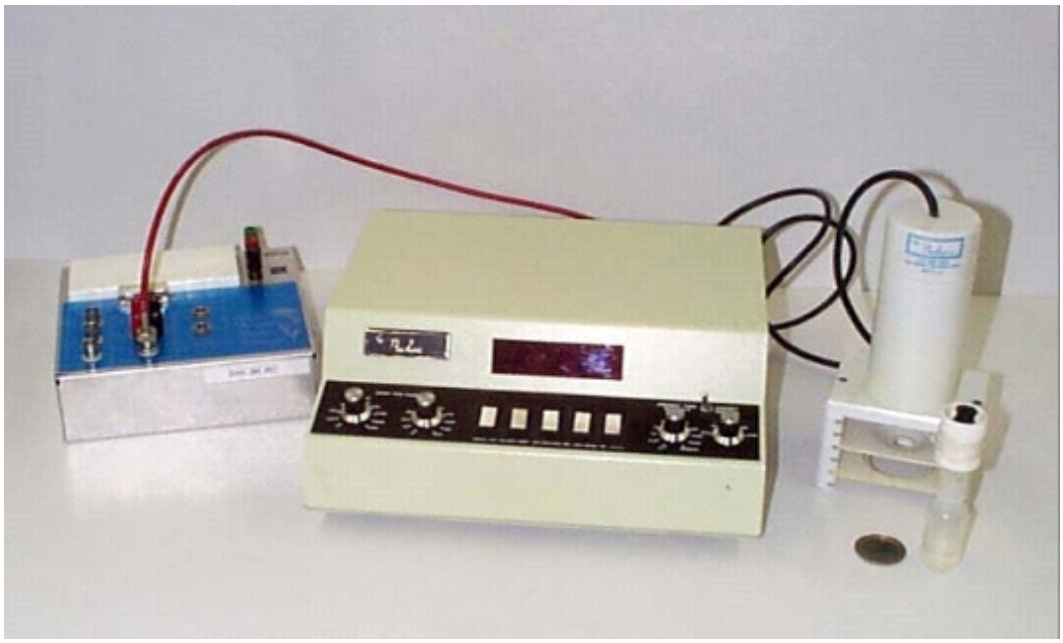

Figure 4.1: Hands-on nuclear decay laboratory setup.

- To measure and analyze the nuclear half-life of an isotope

At the University of Tennessee, an isotope of Barium-137m is used as the radioactive source. The students are instructed to measure the activity of the source as a function of time by counting the number of γ-rays detected over small time intervals. Several factors affect the total number of γ-rays detected, such as detector size and efficiency and background radiation.

When developing an online laboratory, we want to enhance the traditional laboratory, not simply recreate it. Several advantages of an online nuclear decay laboratory are listed below:

- *Reaching a broader target audience*: Safety restrictions are no longer applicable given that students will not be manipulating radioactive sources. In the actual hands-on laboratory, students are told to be careful with the

isotope samples and to avoid any skin contact.

- *Flexibility in choosing a radioactive γ-emitter source*: The decay rate is no longer an issue because the only time constraint is determined by the simulation speed. In addition, sources can be used that would be too dangerous for students to handle in the laboratory.

- *Flexibility in choosing the experimental parameters*: Since experimental data are generated quickly, the only time constraint is determined by the simulation speed. Students have more time to learn about the decay process by varying the experiment parameters and analyzing a larger number of radioactive sources.

- *Teaching Flexibility*: Professors will have the freedom to decide which radioactive source is most appropriate for their class, since the experimental time and safety concerns are no longer an issue. In addition, professors can approach things from a different angle. For example, students can be instructed to measure the half-life of an unknown source and identify the source based on their measurements.

The educational goal of this online laboratory is to teach students how to measure and analyze the decay of several radioactive sources. Students will be able to manipulate the total length of time of the experiment and the length of the short time intervals during which γ-rays are counted. By doing so, students will gain a comprehensive understanding of radioactive decay based on observing the decay of not just one source, but of multiple sources. They will also be able to see how their results depend on the choice of experimental parameters and learn about the importance of choosing appropriate experimental parameters. By determining the decay properties of an unknown decay source and then identifying the source,

students can verify their understanding of the decay process.

4.1 Theory Behind the Nuclear Decay Experiment

A key concept in the nuclear decay theory is the amount of time required for the original activity of a radioactive sample to reduce by half. This time is referred to as the half-life, τ. The following is a mathematical explanation.

Given a radioactive sample containing N radioactive nuclei, we can express the decay rate [21] as follows:

$$R = -\frac{dN}{dt} = \lambda N \tag{4.1}$$

Here R is the decay rate, t is time, and λ is the decay constant. Next, we integrate equation 4.1, from $t = 0$ to an arbitrary time t. This yields:

$$N(t) = N_i e^{-\lambda t} \tag{4.2}$$

Here N_i is the initial number of radioactive nuclei. Recall that the half-life, τ, of a sample is defined as the time required for half of the initial radioactive nuclei to decay. Therefore, by setting $N(t) = \frac{N_i}{2}$ in equation 4.2, we can express the half-life as follows.

$$\tau = \frac{ln(2)}{\lambda} \tag{4.3}$$

Notice that the half-life is dependent on the decay constant.

4.2 Java Implementation

The first step in developing a simulation of nuclear decay is to consider the variables and the level of accuracy required to accomplish the desired educational

goal. With this laboratory, we wish to demonstrate that:

- The nuclear decay process is probabilistic and time independent
- The number of decays as a function of time follows an exponential function

Next, we need to consider the accuracy of the simulation. In this case, we have listed several factors to consider:

- We want to correctly simulate particle decay
- We want to correctly include background noise
- The contribution of the background noise to the data will depend on the distance between the source and the detector

Now that we have determined the primary factors needed to create a realistic online nuclear decay laboratory, we need to consider how to implement them. The following is a detailed description as to how each factor was implemented using Java.

4.2.1 Theoretical Nuclear Decay

To simulate nuclear decay, we have implemented the inverse transformation method. Following the steps provided in section 2.1.1, we finish with the randomly generated time variable given below:

$$t = -\tau ln(r) \tag{4.4}$$

Here t is the generated time, τ is the sample's half-life [22], and r is the randomly generated number. Below is a sample of the Java code from the DecayProcess.class that implements the inverse transformation method to randomly generate time variables.

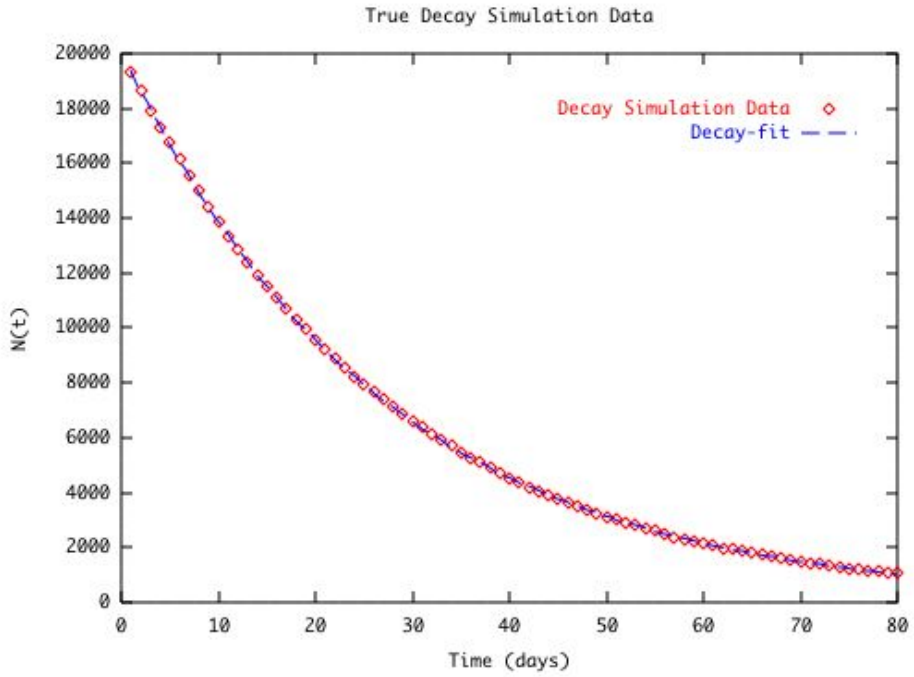

Figure 4.2: Randomly generated nuclear decay events with $\tau = 27$ days.

```
for (int count = 0; count < atoms; count++) {
            double randomizer = Math.log(Math.random());
            timeArray[count] = -randomizer/lamda;
}
```

To produce figure 4.2, we simulated twenty thousand nuclear decay events and compared the distribution of the events with the theoretical distribution.

The results of the fit can be found in table 4.1. The complete source code is available in appendix A.1. As we can see, the simulation results agree with the theoretical model.

Table 4.1: Comparison between the nuclear decay simulation data of Currium 240 with $\tau = 27$ days and the theoretical distribution.

Variable	Theoretical Value	Fitted Value	% Error
τ	27	26.8994±0.01844	0.0595

4.2.2 Background Noise and Detector Location

When we conduct an experiment in the laboratory, we always have background noise. Background radiation may come from several sources, such as natural background radiation (elements in the Earth) and Cosmic ray background radiation [23]. If we are lucky, we can neglect such background noise. However, for this laboratory experiment, background noise can make a significant contribution to the total number of measured γ-rays. In order to simulate the background noise, we used the smearing method discussed in section 2.1.3:

$$\mu \pm \sigma g \tag{4.5}$$

Here μ is the average background noise, σ is the standard deviation, and g is the random sample from the Gaussian distribution.

Here is an exert from the Java code DecayProcess.class, implementing the smearing method to generate the background noise.

```
for (int i = 0; i < bins ; i++) {
     int background = (int) Math.round(Gaussian.getRandomNumber(mu,sigma));
     backgroundArray[i] = background;
}
```

To produce figure 4.3, we have simulated one hundred thousand randomly generated background decay events and compared the events with the theoretical

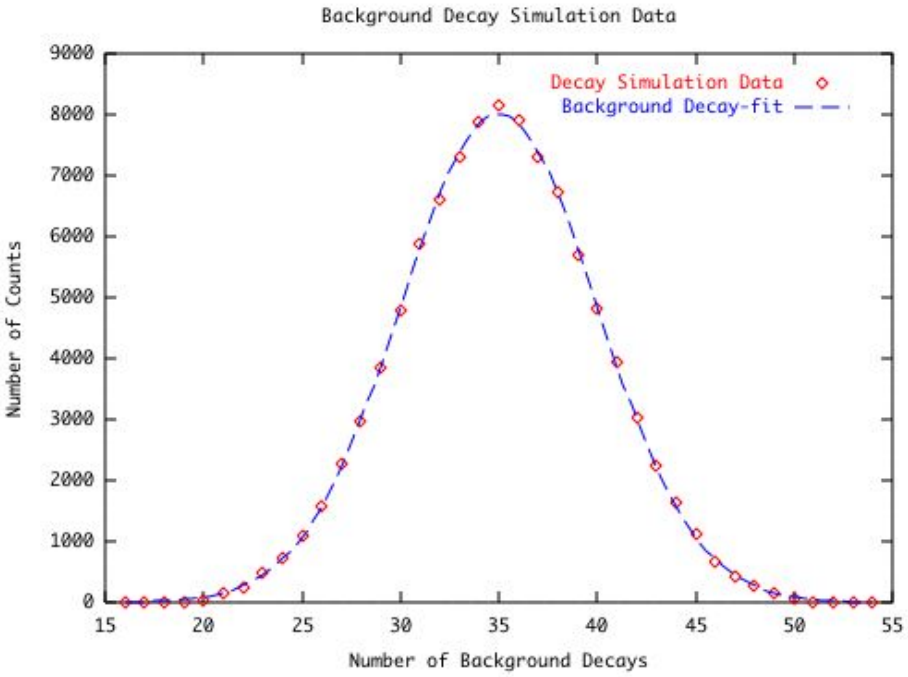

Figure 4.3: Randomly generated background decay events with $\mu = 35$ and $\sigma = 5$.

distribution. The source code is available in appendix A.1. The results of the fit can be found in table 4.2. As we can see, the simulation results agree with the theoretical model.

As mentioned above, to produce an accurate simulation of the nuclear decay experiment conducted in a laboratory environment, it is crucial to consider the location of the detector. In the laboratory, background decays are independent of the detector location. However, we can clearly see that if we move the source away from the detector, the number of decays counted will diminish because the solid angle sustained by the source at the detector will shrink. In the laboratory,

Table 4.2: Comparison between background decay simulation data and the Gaussian distribution with $\mu = 35$ and $\sigma = 5$.

Variable	Theoretical Value	Fitted Value	% Error
μ	35	35.0067±0.01433	0.0601
σ	5	4.98912±0.01433	0.5042

students will have a choice as to the location of the source with respect to the detector. Depending on the location chosen by the student, the simulation will apply a scaling factor and generate fewer real decay events for the same number of background events. This is important when the number of true nuclear decay events approaches the number of background decay events, given that the detector does not distinguish between the types of decays. Once we have randomly generated decay times, we need to create a binning system that counts the number of true nuclear decays and background decays that occurred during an allocated time interval. The following is a segment of the Java code DecayProcess.class showing the binning process.

```
if (i*time_intervals<=timeArray[count] &&
    timeArray[count]<=i*time_intervals+time_intervals) {
    decays++;
    decayArray[i] = decays;
    atomArray[i] = atoms-decayArray[i];
    atomArray[i] = atomArray[i];
    }
}
```

See appendix A.1 for the complete source code for the DecayProcess.class.

4.3 Laboratory GUI

The next step in creating the online nuclear decay laboratory is to create the laboratory GUI. As discussed in section 3.3.1, the GUI has two major components. The first component is the laboratory specific GUI option panel, designed to control the parameters of the experiment. We have identified the following parameters that will be used in our Java code.

- *sourceItem*: The type of radioactive source
- *totalTime*: The total time to run the experiment
- *timeInt*: The time intervals between decay count measurements
- *sampleLocation*: The location of the radioactive source in respect to the detector

The following is a sample of the Java code from the DecayLabGUI.class used to develop the laboratory specific GUI option panel.

```
private static JPanel createDecayOptionPanel() {
    JPanel pane = new JPanel();
    pane.setBorder(BorderFactory.createCompoundBorder(
                    new TitledBorder(new EtchedBorder(), "Simulation Option"),
                    new BevelBorder(BevelBorder.LOWERED)));

        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        pane.setLayout(gridbag);

        //Source Type Label
        buildConstraints(constraints, 0, 0, 1, 1, 30, 30);
        constraints.fill = GridBagConstraints.NONE;
```

```
        constraints.anchor = GridBagConstraints.WEST;
        JLabel sourceType = new JLabel("Source Type: ", JLabel.LEFT);
        gridbag.setConstraints(sourceType, constraints);
        pane.add(sourceType);

         //Source Type Option List
        buildConstraints(constraints, 1, 0, 1, 1, 30, 30);
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.EAST;
        sourceItem.addItem(radon);
        sourceItem.addItem(radium);
        sourceItem.addItem(thorium);
        sourceItem.addItem(currium);
        sourceItem.addItem(unknown);
        gridbag.setConstraints(sourceItem, constraints);
        pane.add(sourceItem);
        .......rest of the code .......
}
```

The second component of the interface is the implementation of the html viewer discussed in section 3.3.1. We combined the laboratory GUI option panel and the html viewer to the create the main panel, which is then presented online using the JApplet class. Please refer to appendix C.2 for the complete Java code of the DecayLabGUI.class and appendix D.1 for the source code of the nuclear decay JApplet.

4.4 Conclusion

We have successfully accomplished our goal for this JLab by implementing various Monte Carlo techniques to simulate physics processes and by developing an online

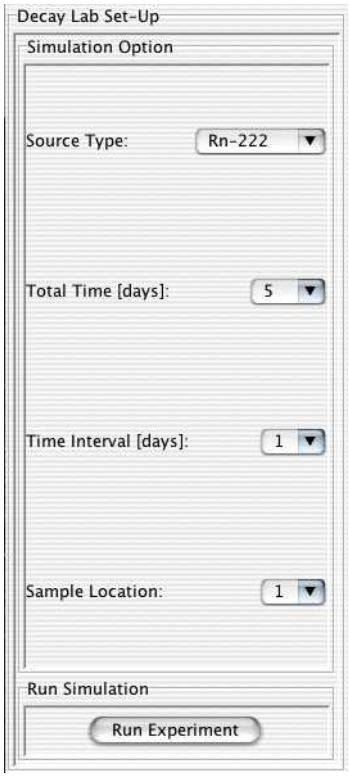

Figure 4.4: Nuclear decay laboratory simulation option panel.

GUI for students to manipulate the laboratory parameters and run the experiment.

We designed a laboratory GUI option panel, as illustrated in figure 4.4, for students to manipulate the parameters of the laboratory. The source code is available in appendix C.2. In addition, we combined both the laboratory GUI option panel and the html viewer into a JApplet, as illustrated in figure 4.5, to present the nuclear decay laboratory on the Internet.

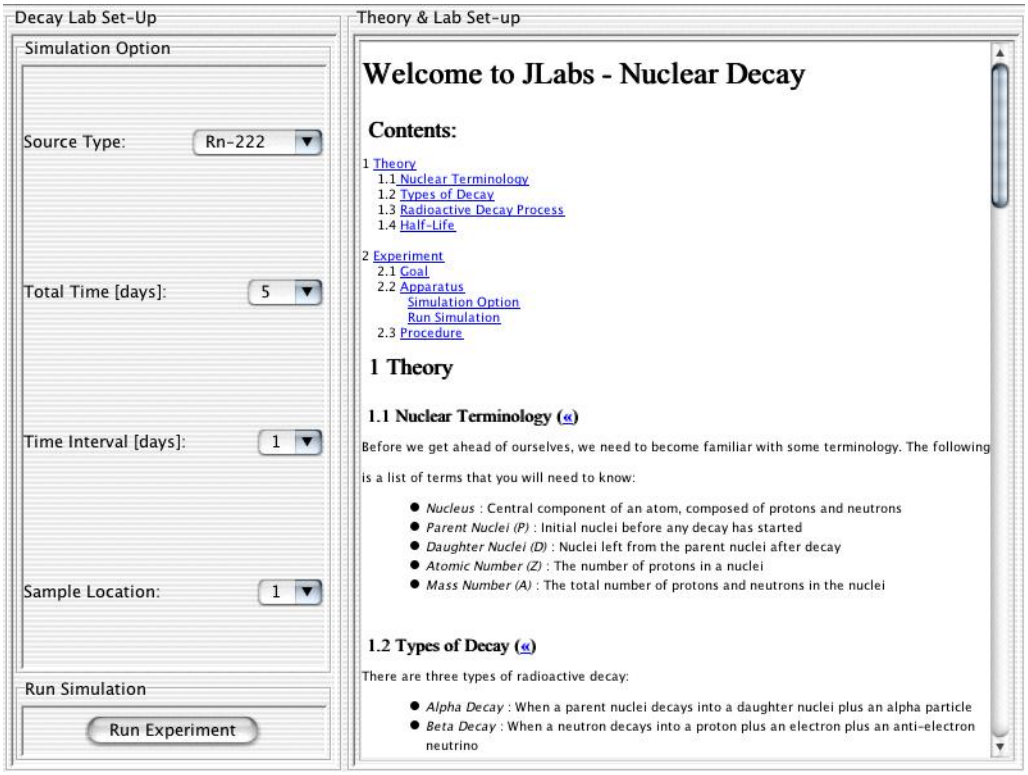

Figure 4.5: JLab - Nuclear Decay Laboratory GUI.

Chapter 5

Java Laboratory 2: Stern-Gerlach Experiment

The Stern-Gerlach online virtual laboratory demonstrates that web-based junior and senior university level laboratories can be successfully created using Monte Carlo techniques. In addition, the laboratory demonstrates that realistic data sets can be created by incorporating Euler's numerical integration technique into the JLab environment.

This JLab represents a variation of a typical third year undergraduate Stern-Gerlach laboratory [24, 25]. The purpose of a traditional hands-on Stern-Gerlach laboratory is to study the deflection of a molecular beam passing through an inhomogeneous magnetic field and to extrapolate the following information from the measurements:

- The most probable deflection of the atoms caused by the non-uniform magnetic field
- The most probable velocity of the atoms within the molecular beam

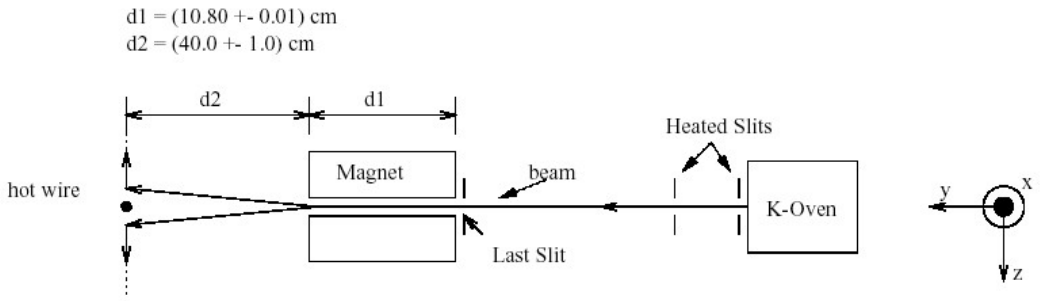

Figure 5.1: Source: Massachusetts Institute of Technology Physics Department, *Junior Physics Laboratory Experiment #18:The Stern-Gerlach Experiment*, 2001

- The magnetic field gradient
- The quantum angular momentum
- The quantum magnetic moment

In the Massachusetts Institute of Technology junior physics laboratory, a potassium beam is passed through the inhomogeneous magnetic field [24]. The students are instructed to measure the deflection of the beam by counting the number of atoms detected as a function of the detector position. The detector is moved in small steps perpendicular to the original direction of the beam.

The experimental setup for a typical Stern-Gerlach experiment is shown in figure 5.1. We will use this diagram as a geometrical guide for our simulation. A beam of potassium atoms emerging from an oven into a field free region is collimated. It is initially moving in the $\hat{y}$-direction. Next, it is passed through a region containing an inhomogeneous magnetic field with $B_z \gg B_x$, i.e $\vec{B}$ is preferentially directed in the $\hat{z}$-direction. The atoms are deflected from their original path and are detected using a hot wire detector. The students measure the distribution of

the deflection angles due to the inhomogeneous magnetic field and determine the information listed above.

The Stern-Gerlach online virtual laboratory has several advantages over its hands-on counterpart. They are listed below:

- *Reaching a broader target audience*: No expensive equipment is needed. Universities with a limited budget will not have to buy expensive equipment.
- *Flexibility in choosing a molecular beam source*: Producing a molecular beam through evaporation for a variety of sources can be technically difficult. In an online virtual laboratory, the speed distribution of atoms emerging from a hot oven is simulated, therefore no oven is needed.
- *Flexibility in choosing the experimental parameters*: Experimental data is generated quickly. Students have more time to learn about the quantum properties of various molecular beams by varying the experimental parameters and by analyzing a larger number of molecular beams.
- *Teaching Flexibility*: Professors will have the freedom to decide which molecular beam source is most appropriate for their class, since the experimental time and equipment limitations are no longer an issue. In addition, professors can approach things from a different angle. For example, students can be instructed to determine the quantum properties of an unknown beam source from their measurements and identify the source based on their analysis.

The educational goal of this online laboratory is for students to learn about the quantization of angular momentum and the Maxwell-Boltzmann distribution. The

online laboratory teaches students how to setup a Stern-Gerlach experiment using various beam sources. Students can manipulate the oven temperature, the magnetic field gradient, the detector position, and the number of atoms in the molecular beam. By manipulating these experimental parameters, students will gain a comprehensive understanding of the Stern-Gerlach experiment. They will understand the quantization of angular momentum based on observing the distribution of deflection for not just one beam source, but for multiple beam sources. They will also be able to see how their results depend on the choice of experimental parameters and learn about the importance of choosing appropriate experimental parameters.

5.1 Theory Behind the Stern-Gerlach Experiment

In classical mechanics, the angular momentum and magnetic moments of an electron orbiting the nucleus of an atom are defined as $\vec{L} = m_e\vec{\omega}r^2$ and $\vec{\mu} = -\frac{e}{2m_e}\vec{L}$, respectively. Here r is the orbital radius, ω is the angular velocity, e is the fundamental charge constant, and m_e is the electron's mass. An atom placed in a magnetic field will have a potential energy $-\vec{\mu}\cdot\vec{B}$. In an inhomogeneous magnetic field with $\frac{\partial B}{\partial z} \neq 0$, the atom will be acted on by a force of $F_z = \mu_z\frac{\partial B_z}{\partial z}$, where μ_z is the projection of the magnetic moment along the $\hat{z}$-direction. This force can have any value between $-|\mu_z|\frac{\partial B_z}{\partial z}$ and $|\mu_z|\frac{\partial B_z}{\partial z}$.

In quantum mechanics, the atom can only exist in discrete states (eigenstates). If we ignore the nuclear magnetic moment, $\vec{I}$, the eigenstates are defined by the square of the angular momentum, $J^2 = j(j+1)\hbar^2$ and by a component of the angu-

lar momentum, for example $J_z = m_j\hbar$. Here j is defined as the angular momentum quantum number with values of integers or half integers and m_j is the magnetic quantum number with the following possible discrete values $-j, -(j-1)...(j-1), j$. We also need to redefine the magnetic moment as $\mu = -g_j\frac{e}{2m_e}\vec{J}$, where g_j is the gyromagnetic ratio of the atomic state. In an inhomogeneous magnetic field with $\frac{\partial B}{\partial z} \neq 0$, the atom will still be acted on by a force of $F_z = \mu_z\frac{\partial B_z}{\partial z}$. However μ_z now takes on only discrete values, $\mu_z = g_j\frac{e}{2m_e}J_z = g_j m_j \mu_B$, where $\mu_B = \frac{e\hbar}{2m_e}$ is the Bohr magneton. Therefore, this force can only have discrete values determined by m_j.

The Stern-Gerlach experiment is designed to probe the core of the quantum mechanical world, specifically the quantum proprieties discussed above. It explores the quantization of the angular momentum and the intrinsic quantum mechanical property we call spin. To observe the quantization of angular momentum, a molecular beam is passed through an inhomogeneous magnetic field and the distribution of the deflection angle is measured and analyzed. Given the type of atoms in the beam, the oven temperature, and the strength of the magnetic field gradient, several properties of the atom can be determined.

5.1.1 Deflection by a Non-Uniform Magnetic Field

The deflection of an atom caused by a non-uniform magnetic field is determined by the magnetic moment μ of the atom and the strength of the magnetic field gradient. The energy of a magnetic moment μ in an applied magnetic field is $-\vec{\mu}\cdot\vec{B}$ [26]. If we introduce a magnetic field perpendicular to the initial direction of the beam and assume that $B_x \cong 0$, then the resulting force on the atom is:

$$\vec{F} = -\vec{\nabla}(\vec{\mu}\cdot\vec{B}) = \mu_B\frac{\partial B}{\partial z}\hat{z} \tag{5.1}$$

While the spin state of an atom is inherently a quantum mechanical property, we can treat the position and momentum of the atom classically because $\Delta p_i \Delta r_i$ does not approach the minimum value required by the uncertainty principle [27]. Therefore, by solving equation 5.1 in terms of the acceleration of the atom, we get:

$$\vec{a} = \frac{\mu_z}{m}\frac{\partial B}{\partial z}\hat{z} \tag{5.2}$$

Given the above acceleration, we can determine the deflection of the atom in the $\hat{z}$-direction as it travels through the magnetic field. If $\frac{\partial B}{\partial z}$ is constant, then:

$$z_1 = \frac{1}{2}at^2 \tag{5.3}$$

If we assume that $v_y \gg v_z$ and the total velocity $v \cong v_y$, then $t = \frac{d_1}{v}$, where d_1 is the distance traveled through the magnetic field. Therefore, the equation for the deflection after passing through the non-uniform magnetic field is:

$$z_1 = \frac{\mu_z}{2mv^2}\frac{\partial B}{\partial z}d_1^2 \tag{5.4}$$

5.1.2 Total Deflection within the Stern-Gerlach Experiment

To determine the total deflection of the molecular beam traversing the apparatus shown in figure 5.1, we need to add the deflection after the passage through the magnetic field z_1 to the deflection z_2 experienced when traveling through the field-free region between the magnet and the detector. Referring to figure 5.1, we have:

$$z_t = z_1 + z_2 = \frac{\mu_z}{2mv^2}\frac{\partial B}{\partial z}(d_1^2 + 2d_1 d_2) \tag{5.5}$$

Here d_1 is the distance traveled through the magnetic field and d_2 is the distance between the magnet and the detector.

5.1.3 Deflection Distribution

The deflection of an atom depends on the speed v with which it emerges from the oven. Equation 5.5 shows that the total deflection is proportional to $\frac{1}{v^2}$. The atoms in the molecular beam emerging from the oven do not have a single fixed velocity but a velocity distribution. According to the Maxwell speed distribution, the number of atoms with speed between v and $v + dv$ is given by [25]:

$$I(v)dv = 2I_0 \frac{v^3}{v_\alpha^4} e^{-(\frac{v}{v_\alpha})^2} dv \tag{5.6}$$

Here $v_\alpha = \sqrt{\frac{2kT}{m}}$ is the most probable velocity from the Maxwell speed distribution and I_0 is the total number of atoms.

Since $z \propto \frac{1}{v^2}$, we have $\frac{dz}{z} = -2\frac{dv}{v}$ and for atoms with a given μ_z, we have $\frac{|z|}{z_\alpha} = (\frac{v_\alpha}{v})^2$, where $z_\alpha = z(v_\alpha)$. Therefore, we can derive the deflection distribution, i.e. the number of atoms arriving at the detector between z and $z + dz$, from equation 5.6.

$$I(z)dz = I_0 \frac{z_\alpha^2}{|z|^3} e^{-(\frac{z_\alpha}{|z|})} dz \tag{5.7}$$

Here z_α is the deflection of the atom with the most probable velocity v_α and is proportional to the $\hat{z}$-component of the magnetic moment μ_z of the atom (see equation 5.4). If the value of μ_z is fixed, then atoms with a Maxwellian speed distribution will be deflected through various angles, with the most probable deflection at $\frac{z_\alpha}{3}$. Here z_α is defined as:

$$z_\alpha = -\frac{\mu_z}{2mv_\alpha^2} \frac{\partial B}{\partial z} (d_1^2 + 2d_1 d_2) \tag{5.8}$$

If μ_z can take on two possible values, say $\mu_z = \pm g_j \frac{\mu_B}{2}$, then the deflection distribution has two maxima at $\pm\frac{z_\alpha}{3}$. A plot of the deflection distribution given by equation 5.7 is shown in figure 5.2.

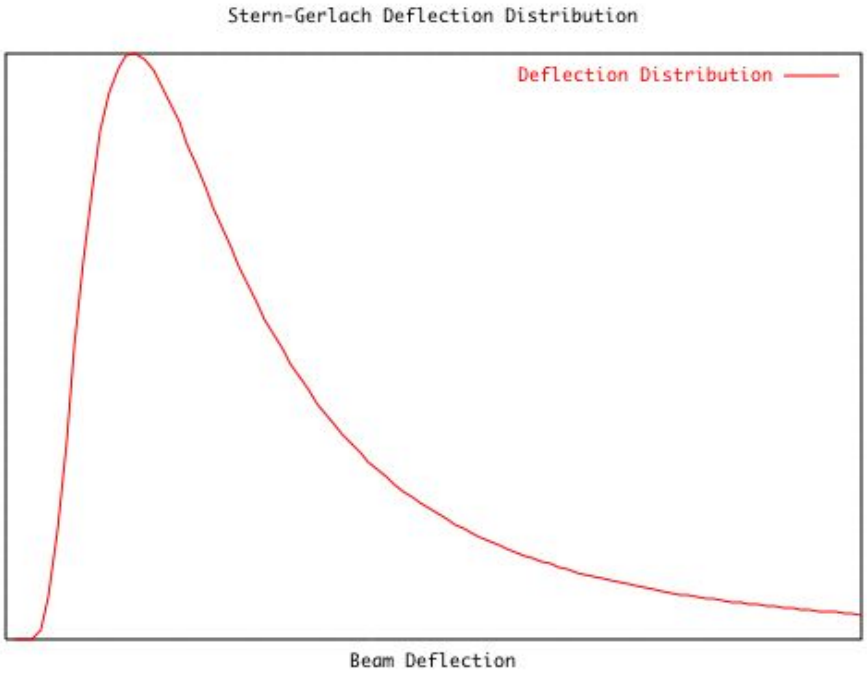

Figure 5.2: Deflection distribution for a single value of μ_z.

5.2 Java Implementation

To create realistic data sets for this JLab, we have to realistically simulate the following:

- The distribution of the quantized magnetic substates of the atom
- The velocity distribution of the emerging molecular beam
- The transport system of the atom

To take advantage of Java's object-oriented property, we have developed individual objects that represent the different physical properties required for this laboratory.

These objects will be introduced in subsequent sections.

5.2.1 Distribution of the Quantized Magnetic Substates of the Atom

In order to simulate atoms emerging from the hot oven in quantized magnetic substates and to access their properties, we have developed a Java object that stores all the properties of each atom.

Atom Object

The Atom class was developed to store crucial physical properties of each atom generated within the simulation. We focused only on the properties that are vital to this experiment, such as mass, position, velocity, acceleration, and spin state. The Atom.class consists of one constructor and one method. The constructor is passed only one argument, the atom's atomic mass. From that argument, the atom is initialized[1] and all of it's initial properties are assigned. One of these properties is the atom's spin state. The getSpin() method was developed to randomly assign the spin state of the atom. For the Stern-Gerlach experiment with potassium atoms, this method has two possible spin states, up or down. Either one is equally likely.

The following segment of the Atom.class is used to generate the spin state of the atom.

```
private static int getSpin(){
    double a = Math.random();
    if(a <= 0.5){
```

[1] An example of how to initialize an Atom object is discussed in section 5.2.3.

Table 5.1: Results of the simulated spin states.

Generated Events	Spin Up	Spin Down	% Difference
1 Million	499550	500450	0.09

```
        return UP;
    } else {
        return DOWN;
    }
}
```

Results from the spin state model can be found in table 5.1. The full Atom.class source code is available in appendix A.3.

5.2.2 Velocity Distribution of the Molecular Beam

The MolecularBeam.class was developed to assign a velocity to the atoms in the beam. This is accomplished by randomly selecting a velocity from the molecular beam's velocity distribution given by equation 5.6. The rejection method is used to assign the velocity to the atom. Results from the molecular beam model are presented in figure 5.3 and table 5.2. The source code is available in appendix A.7.

Table 5.2: Results of v_α from simulated data using potassium atoms at 388.15K.

Variable	Theoretical Value	Fitted Value	% Error
v_α	406.088	406.396±0.114	0.1380

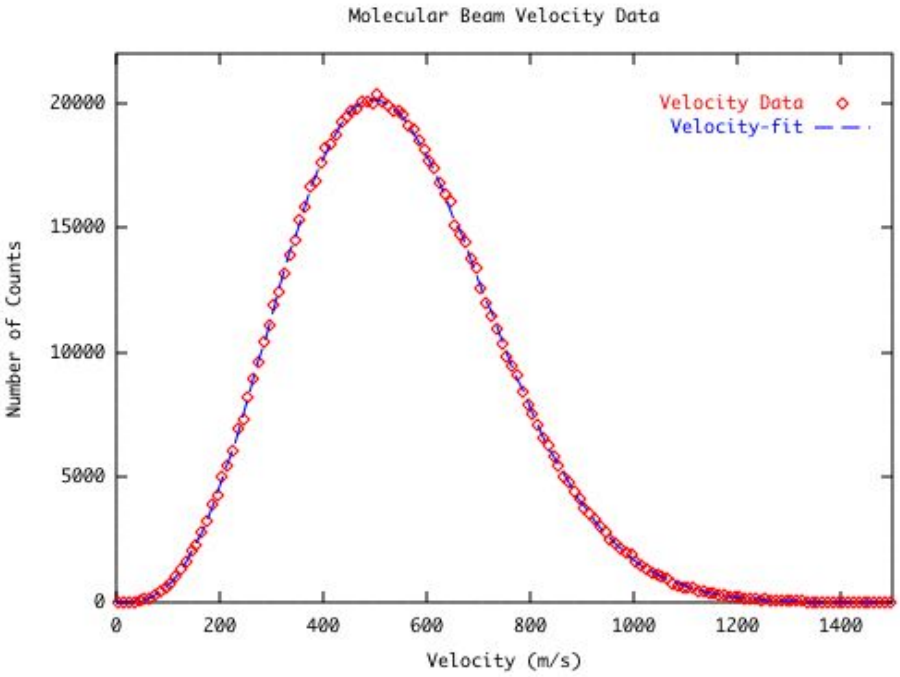

Figure 5.3: Velocity distribution of potassium atoms at 388.15K.

5.2.3 Transport System

To simulate the transport of the atoms from the source to the detector, we developed the SternGerlachSimulation.class and the field and detector objects.

The Field.class was developed to create the non-uniform magnetic field through which the atoms travel. It was designed to create a generic static field, whether it is a gravitational, electric or magnetic field. The field object is designed to create a three dimensional static field. The Field.class has only one constructor which takes no arguments. The source code is available in appendix A.4.

Similar to the Field.class, the detector class has only one constructor which takes no arguments. For the purpose of this simulation, the detector is only a geometrical identity and not a recreation of a real detector. Further development could include various types of real detectors, such as photo-multiplier tubes and their properties, i.e. detection efficiency and noise. The source code for the detector class is available in appendix A.5.

The SternGerlachSimulation object controls all interactions between various objects. The constructor takes seven arguments as listed below:

- getNumber_atoms: Sets the number of events that will be generated
- getAtomic_mass: Sets the atomic mass of the beam source
- getTemp: Sets the temperature of the oven
- getBField: Sets the strength of the magnetic field gradient
- getDSLocation: Sets the detector's initial location on the $\hat{z}$-axis
- getIncrementSize: Sets the distance per increment through which the detector is moved
- getNumberOfIncrements: Sets the total number of increments

The following is a list of the primary methods used within the SternGerlachSimulation object to create the simulation model:

- detectorOn()
- fieldsOn()

- beamOn()
- stepSimulation()
- checkAtomDetected()

The following is the code for SternGerlachSimulation.class constructor used to create the Stern-Gerlach experiment:

```
public SternGerlachSimulation(int getNumber_atoms, double getAtomic_mass,
        double getTemp, double getBField, double getDSLocation,
        double getIncrementSize, int getNumberOfIncrements){
        this.number_atoms = getNumber_atoms;
        this.atomic_mass = getAtomic_mass;
        this.bField = getBField;
        this.yDetectorLocation = getDSLocation;
        this.temp = getTemp;
        this.incrementSize = getIncrementSize;
        this.numberOfIncrements = getNumberOfIncrements;

        int count = 0;
        double ctol = 0.001;  // tolerance [meter]
        deflection = new double[number_atoms];
        countArray = new int[numberOfIncrements];

        detectorOn();
        fieldsOn();
        for(int i = 0; i < number_atoms; i++){
            beamOn();
            dt = ctol/source.vVelocity.y;

            while(source.vPosition.y < wire.vLocation.y + ctol){
                stepSimulation(dt);
```

```
            }
            //Store deflection position of each atom
            deflection[i] = source.vPosition.z;
        }
        //Bin the location of the atoms in respect to the detector location
        for(int j = 0; j < numberOfIncrements; j++){
            wire.vLocation.z = zDetectorLocation+j*incrementSize;
            for(int i = 0; i < number_atoms; i++){
                source.vPosition.y = deflection[i];
                count += checkAtomDetected(source, wire);
            }
            countArray[j] = count;
            count = 0;
        }
}
```

The constructor starts by assigning the arguments' numerical values to seven variables within the simulation. These numerical values are assigned by the end-user via the laboratory GUI option panel. The *count* variable, initially set to zero, is used to keep track of the number of atoms that have hit the detector at each location. The *ctol* variable is used to determine the level of precision of the simulation. It indicates the distance an atom can travel before the program checks if the atom's physical environment has changed.

Once the variables have been defined, the constructor calls on the detectorOn() and fieldsOn() methods to initialize the experiment's detector and field objects, respectively. Next, the simulation starts a loop that initializes an atom by calling the beamOn() method and assigns it an initial velocity. Given the initial velocity and the defined variable *ctol*, the time increment for the Euler's numerical integration technique is determined.

Next, the simulation starts a secondary loop that verifies if the atom is within the experiment's geometry. If the atom is in the experiment's geometry, the stepSimulation() method is applied, otherwise the loop is terminated. The secondary loop will continue until the atom is out of the experiment's geometry, at which point the atom's position is registered using the $deflection[i]$ array. Once the atom's position is registered, the first loop will restart, initializing a new atom until the specified total number of atoms has been generated.

Once all the atoms have been generated, a binning procedure begins. The binning procedure goes through all possible detector locations and determines the number of atoms detected at each location. The procedure starts an initial loop which sets the detector position using the $zDetectorLocation$ and $incrementSize$ variables. Next, a secondary loop is initialized to compare the arrival position of every atom with the detector's position and collection width using the checkAtomDetected() method. Once the arrival position of all the atoms has been checked and the atoms hitting the detector at that position have been counted, the first loop moves the detector to a new position and restarts the second loop. The first loop is finally terminated when all possible detector positions have been checked.

The following sections will provide a detailed account as to how each method works.

detectorOn() Method

The detectorOn() method is used to initialize all the detectors that will be used in the experiment. The following code is an example of the detectorOn() method:

```
static void detectorOn(){
        wire = new Detector();
        wire.vLocation = new Vector(0, 0.608, 0.0056);
        wire.vGeometry = new Vector(0 , 0, 1.27E-4);
}
```

In order to use the detectorOn() method, we need to create a detector object. In the sample code, this is done by creating the detector object called wire. Next, we assign the location and geometry to the wire detector using the Vector class.

fieldsOn() Method

The fieldsOn() method is used to initialize all the fields in the experiment. The following code is an example of the fieldsOn() method:

```
static void fieldsOn(){
        non_uniform = new Field();
        non_uniform.vStartPosition = new Vector(0, 0.1, 0);
        non_uniform.vEndPosition = new Vector(0, 0.208, 0);
        non_uniform.vForce = new Vector(0, 0, 9.27E-22);
}
```

To use the fieldOn() method, we initialize a Field object. In the above code, this is accomplished by creating the Field object non_uniform. Next, we assign the non_uniform object a starting location, geometry, and force using the Vector class.

beamOn() Method

The beamOn() method is used to simulate the molecular beam in the experiment. The molecular beam was developed based on the following assumptions:

- there is no interaction between atoms in the beam

- the beam has no angular divergence

The following code is an example of the beamOn() method:

```
static void beamOn(){
        potassium = new Atom(39.1);
        potassium.vVelocity.y = MolecularBeam.getRandomVelocity(
        388.15, potassium);
}
```

In this example, we have initialized an Atom object named potassium and assigned it a velocity in the $\hat{y}$-direction by using the getRandomVelocity() method from the MolecularBeam class.

stepSimulation() Method

The stepSimulation() method is used to determine the path of the atom. This method is passed one argument, the time step used in Euler's numerical integration technique. When the stepSimulation() method is called, it first updates the position of the atom using the UpdateAtom() method. Once the atom's new position is determined, the stepSimulation() method calls the checkAtomStatus() method to verify if the atom has entered a Field object, and to determine if the equations of motion have changed. The following is the code for the method.

```
static void stepSimulation(double getDeltaT){
        UpdateAtom(potassium, getDeltaT);
        checkAtomStatus(potassium, non_uniform);
}
```

A detailed explanation of the UpdateAtom() and checkAtomStatus() methods is provided below.

UpdateAtom() Method

The UpdateAtom() method applies Euler's numerical integration technique to determine the new location of the atom.

```
static void UpdateAtom(Atom atom, double dt){
        Vector newVelocity = new Vector();
        Vector newPosition = new Vector();

        newVelocity = Vector.vPlus(atom.vVelocity,
                            Vector.vMultiply(atom.vAcceleration, dt));
        newPosition = Vector.vPlus(atom.vPosition,
                            Vector.vMultiply(atom.vVelocity, dt));

        atom.vVelocity = newVelocity;
        atom.vPosition = newPosition;
}
```

checkAtomStatus() Method

The checkAtomStatus() method verifies if the atom has entered the geometry of the field. If the atom is in the field object's geometry, a force acts on the atom and it accelerates.

```
static void checkAtomStatus(Atom atom, Field field){
        if(atom.vPosition.y > field.vLocation.y
           && atom.vPosition.y < field.vLocation.y + field.vGeometry.y)
       {
            Vector bForce = field.vForce;
            atom.vAcceleration = Vector.vMultiply(
                     Vector.vDivide(bForce, atom.mass), atom.spin);
        } else {
            atom.vAcceleration.z = 0;
        }
```

```
}
```

checkAtomDetected() Method

The checkAtomDetected() method is used to determine if the atom has entered the geometrical space of the detector. If this is the case, then the method returns a value of one, otherwise it will return a value of zero. The following is the code for the method:

```
static int checkAtomDetected(Atom atom, Detector detector){
        int retvalue;
        if(atom.vPosition.y >= detector.vLocation.y &&
           atom.vPosition.y <= detector.vLocation.y + detector.vGeometry.y)
        {
            retvalue = 1;
        } else {
            retvalue = 0;
        }
        return retvalue;
}
```

5.2.4 Simulation Results

To check if our simulation produces realistic results, we have to compare our simulation results to the theoretical predictions discussed in section 5.1.3.

In our simulation, we used potassium ^{39}K as the atomic source for the beam. The electronic structure of ^{39}K in it's ground state is given below [28]:

$$^{39}K = 1s^2 2s^2 2p^6 3s^2 3p^6 4s \tag{5.9}$$

Table 5.3: Quantum numbers and electron orbital designation for ^{39}K in it's ground state.

n	l	m_l	Electron Orbital Designation
1	0	0	$1s^2$
2	0	0	$2s^2$
	1	-1, 0, 1	$2p^6$
3	0	0	$3s^2$
	1	-1, 0, 1	$3p^6$
	2	-2, -1, 0, 1, 2	N/A
4	0	0	4s

Table 5.3 lists the quantum numbers for each orbital and for the electron orbital designation. The Russel-Saunders notation for the electronic ground state of ^{39}K is $^2S_{1/2}$ [28]. We can infer that the total angular momentum due to the electronic structure is $\vec{J} = \hbar/2$. For the simulation, we assume that we can neglect the nuclear magnetic moment $\vec{I}$ since $\vec{I}$ and $\vec{J}$ are "decoupled" when a strong field is applied to the atom. Therefore, the effect of $\vec{I}$ is not noticeable since $\frac{\mu_N}{\mu_B} \propto \frac{m_e}{m_p} \approx \frac{1}{1836}$ [29]. Hence, ^{39}K has approximately the same magnetic moment as the electrons intrinsic magnetic moment and $\mu_z \cong \pm\mu_B$. We can therefore expect the deflection distribution given by equation 5.7. Table 5.4 compares the

Table 5.4: Results of z_α from the simulation data using a potassium beam at 388.15 K.

Variable	Theoretical Value	Fitted Value	% Error
z_α	0.004244	0.004321±4.083E-6	1.918

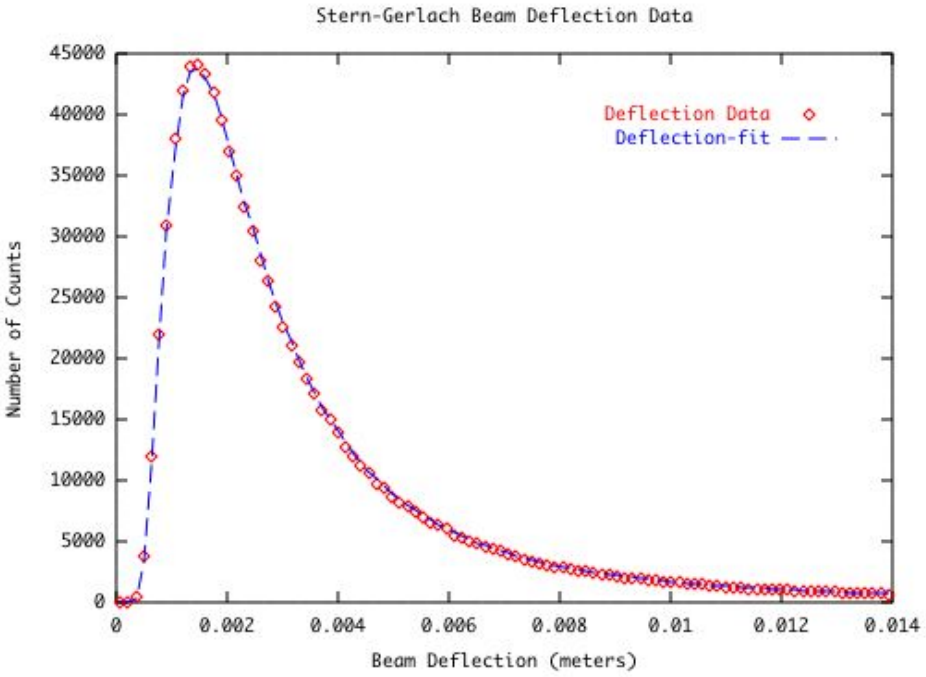

Figure 5.4: Beam deflection distribution of potassium at 388.15K.

predicted value of z_α with the value extracted from the simulation data. Figure 5.4 shows the deflection distribution produced by the simulation and compares it with equation 5.7. As we can see, the simulation results agree with the theoretical model.

5.3 Laboratory GUI

The next step in creating the online Stern-Gerlach laboratory is to create the laboratory GUI. As discussed in section 3.3.1, the GUI has two major components. The first component is the laboratory specific GUI option panel, designed to

control the parameters of the experiment. The following parameters are used in our Java code.

- *sourceItem:* The molecular beam source type
- *temperature:* The temperature assigned to the oven
- *magneticField:* The strength of the magnetic field gradient
- *detectorStartLocation:* The starting location of the detector
- *incrementSize:* The distance per increment the detector will move
- *numberOfIncrement:* The number of increments the detector will move
- *numberOfEvent:* The number of atoms generated in the simulation

The following is a sample of the Java code from the sgLabGUI.class used to develop the laboratory specific GUI option panel.

```
private static JPanel createSternGerlachOptionPanel() {
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Simulation Option"),
                new BevelBorder(BevelBorder.LOWERED)));

    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    pane.setLayout(gridbag);

    //Source Type Label
    buildConstraints(constraints, 0, 0, 1, 1, 30, 30);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.WEST;
```

```
        JLabel sourceType = new JLabel("Source Type: ", JLabel.LEFT);
        gridbag.setConstraints(sourceType, constraints);
        pane.add(sourceType);

         //Source Type Option List
        buildConstraints(constraints, 1, 0, 1, 1, 30, 30);
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.EAST;
        sourceItem.addItem(potassium);  // a_mass = 39.1
        sourceItem.addItem(silver);  // a_mass = 108
        gridbag.setConstraints(sourceItem, constraints);
        pane.add(sourceItem);
        .......rest of the code .......
}
```

The second component of the interface is the implementation of the html viewer discussed in section 3.3.1. We combine the laboratory GUI option panel and the html viewer to the create the main panel, which is then presented online using the JApplet class.

Please refer to appendix C.3 for the complete Java code of the SternGerlach-LabGUI.class and appendix D.2 for the source code of the Stern-Gerlach JApplet.

5.4 Conclusion

The Stern-Gerlach JLab is a proof of concept, demonstrating that online advanced physics laboratories can successfully be created. We have accomplished our goal for this JLab by implementing various Monte Carlo techniques and Euler's method to realistically simulate the physics processes involved in the experiment.

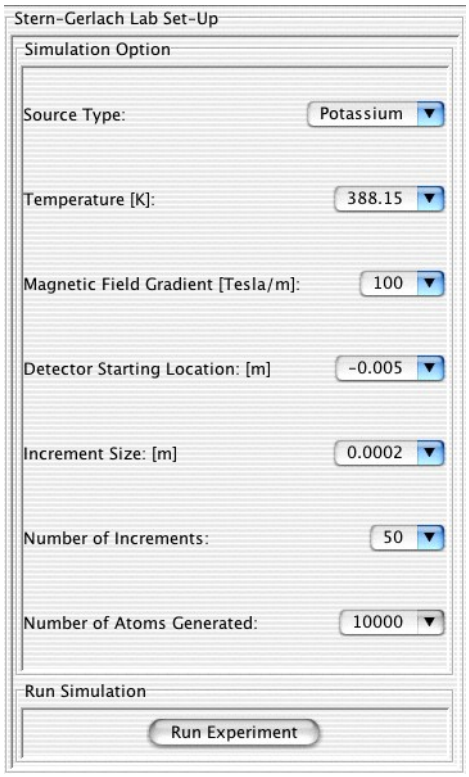

Figure 5.5: Stern-Gerlach laboratory simulation option panel.

We designed a laboratory GUI option panel, as illustrated in figure 5.5, for students to manipulate the experimental parameters of the laboratory and run the experiment. The source code is available in appendix C.3. We have not yet developed the HTML code, presenting background material and instructions. Depending on the instructional environment in which this JLab is used, this material could focus on the experiment or could present extensive background information concerning angular momentum, spin states, etc.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have successfully developed a robust and sophisticated online physics laboratory environment. This environment can handle large data sets and create realistic experimental results by applying Monte Carlo and numerical integration techniques. In this environment, we have developed two JLabs designed to help increase student's understanding of physics. These JLabs are:

- *Online Virtual Nuclear Decay Laboratory*: A functional interactive online laboratory, applying Monte Carlo techniques.
- *Online Virtual Stern-Gerlach Laboratory*: A proof of concept, demonstrating that online advanced physics laboratories can successfully be created using both Monte Carlo and numerical integration techniques.

These JLabs teach students how to setup experiments, manipulate experimental parameters, take data, and use various analysis techniques. Students gain a comprehensive understanding of various physical processes.

Table 6.1: Comparison between Java and Flash 5.

Environment	Cross-Platform	Generating Simulations	Expandability
Java	Yes	Yes	Yes
Flash 5	Yes	Limited	Yes

To achieve the goals of this thesis, we devoted some time to understanding the advantages and limitations of both the Flash 5 and Java development environments. In considering both environments, we needed to make sure that the following objectives were met:

- cross-platform
- generation of true simulation models
- expandable

We developed a nuclear decay laboratory using both Flash 5 and Java. We decided to use Java to further develop the online laboratories, given the limited simulation capabilities of Flash 5. Table 6.1 is a summary of the characteristics of both environments with respect to the goals stated above. For more information on the development environments, please refer to chapter 3.

In the process of developing the JLab environment, we created a Java physics library that can be used for future development of online laboratories. The following is a list of the classes within the Java physics library:

- mjava.mathematics.Gaussian.java
- mjava.mathematics.Vector.java

- mjava.physics.Atom.java
- mjava.physics.Field.java
- mjava.physics.Detector.java
- mjava.physics.MaxwellDistribution.java
- mjava.physics.DecayProcess.java
- mjava.physics.MolecularBeam.java
- mjava.physics.SternGerlachSimulation.java
- mjava.io.Print.java
- mjava.gui.HTMLViewer.java
- mjava.gui.DecayLabGUI.java
- mjava.gui.SternGerlachLabGUI.java

Given that these classes have been designed focusing on the online laboratories created within this thesis, some minor modifications may be required if they were to be used for additional online laboratories.

6.2 Final Thoughts and Future Possibilities

The development within this thesis has focused exclusively on simulation techniques required to create a university level online laboratory. It is the authors belief that online laboratories should simulate the physics processes and not just simply smear published results.

6.2.1 Major Enhancements

To further develop the JLab online learning environment, there are four major enhancements that should be implemented:

- Combining Flash 5 and Java
- Icon-Oriented Modeling System
- Professor GUI Interface
- Analysis & Visualization

Combining Flash 5 and Java

Flash 5 could be used to introduce students to the experiment, to present the theory, and even to allow students to manipulate some of the experimental parameters. Students could then link to the JLab when they are ready to run the experiment. This would reduce the time it takes to develop a new online laboratory, since Flash 5 requires much less programming.

Icon-Oriented Modeling System

The present JLabs focus on having the student change the input parameters and analyze the output data. The next step in developing a realistic laboratory environment is to have the students setup the laboratory. This could be done by having icons represent the individual instruments used in the experiment. The student can then manipulate the instruments by moving them to the desired locations, turning them on or off when appropriate, and setting up the inital values of the instruments. Icon-oriented modeling systems have already been developed and promising results have been reported [30].

Professor GUI Interface

Further development of JLabs should consider professors' needs to modify the laboratory. The present JLab environment does not let professors expand the choices given to the students without having to manipulate the source code. For example, a professor with a particular interest in the decay rate of a specific element should be able to easily incorporate that source into the laboratory. A GUI interface should be developed to allow professors to expand the individual JLab.

Analysis & Visualization

Another worth while addition to the JLab environment is a self-contained analysis and visualization system. The analysis portion should allow the students to manipulate the data directly online. The visualization portion should allow the students to observe some of the physical processes in real time or in time lapse to help them understand how the physics works. A wonderful example of what we can offer students in terms of analysis and visualization is CERN's online project "Hands on CERN" [31]. The "Hands on CERN" web site teaches the public about particle physics, the methods used to probe the secrets of the Universe, and allows students and teachers to analyze new data sets from CERN.

6.2.2 Minor Enhancements

In addition to the major enhancements listed above, there are a few minor changes that would add functionality to the JLab environment.

Extensible Markup Language & Mathematical Markup Language

In the future, we should try and add more functionality to the html viewer by implementing Extensible Markup Language (XML) [32]. The release of Java 1.4 added functionality to the java packages such that we can implement XML into Java applets [10]. There are numerous benefits in implementing XML such as: the ability to create a standard representation of the procedures and theories; and the ability to present mathematical equations using the XML application Mathematical Markup Language (MathML) [33].

Developing New Simulation Techniques

The present development of the JLab environment uses simple simulation techniques. Development of more sophisticated simulation models may require the use of different techniques such as the Runge-Kutta method or various interpolation methods.

Binning System

In this thesis, a binning system was developed for each JLab. To facilitate the development of future JLabs, a generic binning system should be developed. This binning system could be called by JLabs simulating various physical processes and could be part of the analysis package.

Bibliography

Bibliography

[1] E. F. Redish J. M. Saul and R. N. Steinberg. On the effectiveness of active-engagement microcomputer-based laboratories. *American Journal of Physics*, 65(1):45–54, 1997.

[2] Pricilla W. Laws. Workshop physics: Replacing lectures with real experience. In *Conference on Computers in Physics Instructions*, pages 22–32. North Carolina State University, Addison-Wesley Publishing Company, August 1988.

[3] L. C. McDermott. How we teach and how students learn – a mismatch? *American Journal of Physics*, 60:295, 1993.

[4] Anne J. Cox and William F. Junkin III. Enhanced student learning in the introductory physics laboratory. *Physics Education*, 37(1):37–44, January 2002.

[5] John Clinch and Kevin Richards. How can the internet be used to enhance the teaching of physics? *Physics Education*, 37(2):109–114, March 2002.

[6] http://www.if.ufrgs.br/∼betz/quantum/SGtext.htm

[7] Marianne Breinig et al. A proposed center of excellence in advanced educational technology, 2001.

[8] Philip R. Bevington and D. Keith Robinson. *Data Reduction and Error Analysis for Physical Sciences*. McGraw-Hill, 2nd edition, 2001.

[9] F. James S. Youssef, R. Cousins. Monte Carlo techniques. *The European Physical Journal C*, 15(1-4):202–204, Feb 2000.

[10] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Inc., 4th edition, 2002.

[11] W. H. Press et al. *Numerical Recipes in C: The Art of Scientific Computing*. Press Syndicate of the Univeristy of Cambridge, 2nd edition, 1992.

[12] David M. Bourg. *Physics for Game Developers*. O'Reilly & Associates, Inc., 1st edition, 2002.

[13] Richard Feynman. *The Feynman Lectures on Physics*, volume 1. Addison-Wesley, 1989.

[14] Mary L. Boas. *Mathematical Methods in the Physical Sciences*. John Wiley & Sons, Inc., 2nd edition, 1983.

[15] Marianne Breinig, Private communication.

[16] Mike Guitry, Private communication.

[17] Jody Bleyle. *Macromedia Flash 5: Using Flash*. Macromedia, Inc., 1st edition, 2000.

[18] Jody Bleyle. *Macromedia Flash 5: ActionScript Reference Guide*. Macromedia, Inc., 1st edition, 2000.

[19] Steven Gutz. *Up to Speed with Swing: User Interfaces with Java Foundation Classes*. ManningPublications Co., 2nd edition, 2000.

[20] Jr. Dean S. Edmonds. *Cioffari's Experiments in College Physics*. D.C. Heath and Company, 8th edition, 1988.

[21] Raymond A. Serway and Jerry S. Faughn. *College Physics*. Saunders College Publishing, 4th edition, 1995.

[22] James William Rohlf. *Modern Physics from α to Z^0*. John Wiley & Sons, Inc., 1st edition, 1994.

[23] A. Fasso R.J. Donahue. Radioactivity and radiation protection. *The European Physical Journal C*, 15(1-4):186–189, March 2000.

[24] Massachusetts Institute of Technology Physics Department. *Junior Physics Laboratory Experiment #18: The Stern-Gerlach Experiment*, 2001.

[25] University of Wisconsin. *Stern-Gerlach: Advanced Laboratory, Physics 407*, 2002.

[26] David J. Griffiths. *Introduction to Electrodynamics*. Prentice Hall, 2nd edition, 1989.

[27] J. J. Sakurai. *Modern Quantum Mechanics*. Addison-Wesley Publishing Company, Inc., 1994.

[28] W.C. Martion. Electronic structure of the elements. *The European Physical Journal C*, 15(1-4):78–79, 1999.

[29] B.N. Taylor P.J. Mohr. Physical constant. *The European Physical Journal C*, 15(1-4):73, 2000.

[30] Horst Schecker. Learning physics by making models. *Physics Education*, 26:102–106, 1993.

[31] K. E. Johanson and T. G. M. Malmgren. Hands on CERN: an educational project on the internet using real high energy particle collisions. *Physics Education*, 34(5):286–293, September 1999.

[32] Tim Bray et al. *Extensible Markup Language (XML)*. W3C, 1st edition, October 2000.

[33] Ron Ausbrooks et al. *Mathematical Markup Language (MathML)*. W3C, 2nd edition, 2001.

Appendix

Appendix A

Physics

A.1 DecayProcess.class

```
public class DecayProcess {

public static double[]
decayProcess(double hf,int total_time, int time_intervals,int sample_location){

        int atoms = 20000;
        int decays = 0;

        double lamda = Math.log(2)/hf;
        int bins = total_time/time_intervals;
        atoms = (int) atoms/((int) (Math.pow(sample_location, 2)));

        // Random decay time generator --> timeArray[count]
        double timeArray[] = new double[atoms];
        for (int count = 0; count < atoms; count++) {
            double randomizer = Math.log(Math.random());
            timeArray[count] = -randomizer/lamda;
```

```
    }

    // Background Noise generator --> backgroundArray[bins]
    int mu = 35;
    double sigma = 5;
    double backgroundArray[] = new double[bins];
    double dataArray[] = new double[bins];
    double atomArray[] = new double[bins];
    double decayArray[] = new double[bins];
    for (int i = 0; i < bins ; i++) {
        int background = (int) Math.round(
             Gaussian.getRandomNumber(mu,sigma));
        backgroundArray[i] = background;

    // Bin loop for timeArray
    for (int count = 0; count < atoms; count++) {
        if (i*time_intervals<=timeArray[count] &&
            timeArray[count]<=i*time_intervals+time_intervals) {
            decays++;
            decayArray[i] = decays;
            atomArray[i] = atoms-decayArray[i];
            atomArray[i] = atomArray[i];
            }
        }
    dataArray[i] = backgroundArray[i]+atomArray[i];
    }
    return dataArray;
}

public static double[] binProcess(int total_time,int time_intervals) {
    int bins = total_time/time_intervals;
    double binArray[] = new double[bins];
```

```
        for(int i = 0; i < bins; i++){
            binArray[i] = (i+1)*time_intervals;
        }
        return binArray;
    }
}
```

A.2 MaxwellDistribution.class

```
public class MaxwellDistribution {

    public static final double k = 8.61739E-5;  // Boltzmann Constant [eV/K]
    public static final double c = 2.9979E8;  // Speed of Light[m/s]
    public static final double kgToeV = (931.5E6)/(1.66E-27);

    //Method used to determine the mean velocity
    public static double getMeanVelocity(double temperature, Atom atom){
        double mass = getMass(atom.mass);
        return c*Math.sqrt((8*k*temperature)/(Math.PI*mass));
    }

    //Method used to determine the most probable velocity
    public static double getMaxVelocity(double temperature, Atom atom){
        double mass = getMass(atom.mass);
        return c*Math.sqrt((2*k*temperature)/(mass));
    }

    //Method used to randomly generate a velocity from the maxwell distribution
    public static double getRandomVelocity(double temperature, Atom atom){
        double good_v = 0;
        double v_max = getMaxVelocity(temperature, atom);
```

```
    double max_distribution =
               maxwell_distribution(temperature, v_max, atom);

    while(good_v == 0){
        double random_distribution = max_distribution*Math.random();
        double random_v = 5*v_max*Math.random();
        double calculated_distribution = maxwell_distribution(
                                  temperature, random_v, atom);

        //Applying ther Rejection Method to generated points
        if (random_distribution < calculated_distribution) {
            good_v = random_v;
        } else {
            good_v = 0;
        }
    }
    return good_v;
}

//Maxwell's Distribution Funtion
private static double maxwell_distribution(
  double temperature, double velocity, Atom atom) {
  double mass = getMass(atom.mass);
  double alpha =
  4*Math.PI*Math.pow(c,1/3)*Math.pow((mass)/(2*Math.PI*k*temperature),3/2);
  double beta = mass/(2*k*temperature*Math.pow(c,2));
  return alpha*Math.pow(velocity,2)*Math.exp(-beta*Math.pow(velocity,2));
}

//Method that convert the mass from [kg]->[eV/(c^2)]
private static double getMass(double massKG){
    return massKG*(kgToeV);
```

```
    }
}
```

A.3 Atom.class

```
public class Atom{

    public static final int UP = 1;  //Spin state
    public static final int DOWN = -1;  //Spin state
    public double atomic_mass;  //Atomic mass of the atom
    public double mass;  //Mass of of the atom [kg]
    public int spin;  //Spin of the atom [UP/DOWN]
    public Vector vPosition = new Vector(); //Position of atom [m]
    public Vector vVelocity = new Vector();  //Velocity of atom [m/s]
    public Vector vAcceleration = new Vector();//Acceleration of atom [m/(s^2)]

    //Constructor
    public Atom(double atomic_mass){
        //set mass of atom
        this.mass = atomic_mass*(1.66E-27);

        //set atoms spin
        this.spin = getSpin();

        //Set Initial Position
        this.vPosition.x = 0;
        this.vPosition.y = 0;
        this.vPosition.z = 0;

        //Set Inital Velocity
        this.vVelocity.x = 0;
        this.vVelocity.y = 0;
```

```
        this.vVelocity.z = 0;

        //Set Initial Acceleration
        this.vAcceleration.x = 0;
        this.vAcceleration.y = 0;
        this.vAcceleration.z = 0;
    }

    //Method that randomly generates the spin state of the
    //atom (limited to UP or DOWN)
    private static int getSpin(){
        double a = Math.random();
        if(a <= 0.5){
            return UP;
        } else {
            return DOWN;
        }
}
```

A.4 Field.class

```
public class Field {

    public Vector vLocation = new Vector(); //Location of the Field  [meter]
    public Vector vGeometry = new Vector(); //Field Geometry[meter]
    public Vector vForce = new Vector();  //Force applied by the Field [N]

    //Constructor
    public Field(){

        //Set the Location of Physics Process
        this.vLocation.x = 0;
```

```
        this.vLocation.y = 0;
        this.vLocation.z = 0;

        //Set Geometry of Physics Process
        this.vGeometry.x = 0;
        this.vGeometry.y = 0;
        this.vGeometry.z = 0;

        //Set Force from the Field
        this.vForce.x = 0;
        this.vForce.y = 0;
        this.vForce.z = 0;
    }
}
```

A.5 Detector.class

```
public class Detector {

    public Vector vLocation = new Vector(); //Location of the Detector [meter]
    public Vector vGeometry = new Vector(); //Detector Geometry [meter]

    //Constructor
    public Detector(){

        //Set location of Detector
        this.vLocation.x = 0;
        this.vLocation.y = 0;
        this.vLocation.z = 0;

        //Set geometry of Detector
        this.vGeometry.x = 0;
```

```
        this.vGeometry.y = 0;
        this.vGeometry.z = 0;
    }
}
```

A.6 SternGerlachSimulation.class

```
public class SternGerlachSimulation {
        static int number_atoms, numberOfIncrements;
        static double atomic_mass, temp, bField,
                            zDetectorLocation, incrementSize;
        static Atom source;
        static Field non_uniform;
        static Detector wire;
        static double dt;
        public static double deflection[];
        public static int countArray[];

    //This constructor has been designed to run the Stern-Gerlach experiment
    public SternGerlachSimulation(int getNumber_atoms, double getAtomic_mass,
              double getTemp, double getBField, double getDSLocation,
              double getIncrementSize, int getNumberOfIncrements){
        this.number_atoms = getNumber_atoms;
        this.atomic_mass = getAtomic_mass;
        this.bField = getBField;
        this.zDetectorLocation = getDSLocation;
        this.temp = getTemp;
        this.incrementSize = getIncrementSize;
        this.numberOfIncrements = getNumberOfIncrements;
        int count = 0;
        double ctol = 0.001;  // tolerance [meter]
        deflection = new double[number_atoms];
```

```
    countArray = new int[numberOfIncrements];

    detectorOn();
    fieldsOn();
    for(int i = 0; i < number_atoms; i++){
        beamOn();
        dt = ctol/source.vVelocity.y;

        while(source.vPosition.y < wire.vLocation.y + ctol){
            stepSimulation(dt);
        }
        //Store deflection position of each atom
        deflection[i] = source.vPosition.z;
    }
    //Bin the location of the atoms in respect to the detector location
    for(int j = 0; j < numberOfIncrements; j++){
        wire.vLocation.z = zDetectorLocation+j*incrementSize;
        for(int i = 0; i < number_atoms; i++){
            source.vPosition.z = deflection[i];
            count += checkAtomDetected(source, wire);
        }
        countArray[j] = count;
        count = 0;
    }
}

static void stepSimulation(double getDt){
    UpdateAtom(source, getDt);
    checkAtomStatus(source, non_uniform);
}

//Method used to create a atom
```

```
static void beamOn(){
    source = new Atom(atomic_mass);
    source.vPosition.z = Gaussian.getRandomNumber(0, 3E-4);
    source.vVelocity.y = MolecularBeam.getRandomVelocity(temp, source);
}

//Method to create a field
static void fieldsOn(){
    non_uniform = new Field();
    non_uniform.vLocation = new Vector(0,0.1,0);
    non_uniform.vGeometry = new Vector(0,0.108, 0);
    non_uniform.vForce = new Vector(0, 0, 9.2731E-24*bField);
}

//Method to create a detector
static void detectorOn(){
    wire = new Detector();
    wire.vLocation = new Vector(0, 0.608, 0);
    wire.vGeometry = new Vector(0, 0, 1.9E-4);
}

//Method used to apply the equations of motion on the atom
static void UpdateAtom(Atom atom, double dt){
    Vector newVelocity = new Vector();
    Vector newPosition = new Vector();

    newVelocity = Vector.vPlus(atom.vVelocity,
                          Vector.vMultiply(atom.vAcceleration, dt));
    newPosition = Vector.vPlus(atom.vPosition,
                          Vector.vMultiply(atom.vVelocity, dt));

    atom.vVelocity = newVelocity;
```

```
        atom.vPosition = newPosition;
    }

    //Method used to determine the status of the atom in regards to the field
    static void checkAtomStatus(Atom atom, Field field){

        if(atom.vPosition.y > field.vLocation.y  &&
           atom.vPosition.y < field.vLocation.y + field.vGeometry.y){
            Vector bForce = field.vForce;
            atom.vAcceleration = Vector.vMultiply(
                  Vector.vDivide(bForce, atom.mass), atom.spin);
        } else {
            atom.vAcceleration.z = 0;
        }
    }

    static int checkAtomDetected(Atom atom, Detector detector){
        int retvalue;
        if(atom.vPosition.z >= detector.vLocation.z &&
           atom.vPosition.z < detector.vLocation.z + detector.vGeometry.z)
        {
            retvalue = 1;
        } else {
            retvalue = 0;
        }
        return retvalue;
    }
}
```

A.7 MolecularBeam.class

```
public class MolecularBeam {
```

```
    //Method used to randomly generate a
    //velocity from the molecular beam distribution
    public static double getRandomVelocity(double temperature, Atom atom){
        double good_v = 0;
        double mv_max = MaxwellDistribution.getMaxVelocity(temperature, atom);
        double v_max = mv_max*Math.sqrt(1.5);
        double max_distribution = beam_distribution(mv_max, v_max);

        while(good_v == 0){
            double random_distribution = max_distribution*Math.random();
            double random_v = 4*v_max*Math.random();
            double calculated_distribution =
                       beam_distribution(mv_max, random_v);

            //Applying ther Rejection Method to generated points
            if (random_distribution < calculated_distribution) {
                good_v = random_v;
            } else {
                good_v = 0;
            }
        }
        return good_v;
    }
    //Molecular Beam Distribution
    private static double beam_distribution(
            double maxwellMPVelocity, double velocity) {
            double alpha =
            2*Math.pow(velocity,3)/Math.pow(maxwellMPVelocity,4);
            double beta = Math.pow(velocity/maxwellMPVelocity,2);
            return alpha*Math.exp(-beta);
    }
}
```

Appendix B

Mathematics

B.1 Gaussian.class

```
public class Gaussian {

    //Method used to randomly generate a number from the Gaussian distribution
    public static double getRandomNumber(double mean, double sigma){
        double calculated_distribution = 0;
        double goodNumber = 0;
        double max_distribution = 1/(sigma*Math.sqrt(2*Math.PI));
        while(goodNumber == 0){
            double random_distribution = max_distribution*Math.random();
            double randomNumber = mean + 3*sigma*(2*Math.random()-1);
            calculated_distribution = gaussian(mean, sigma, randomNumber);

            //Applying ther Rejection Method to generated points
            if (random_distribution < calculated_distribution) {
                goodNumber = randomNumber;
            } else {
                goodNumber = 0;
```

```
            }
        }
        return goodNumber;
    }
    //Gaussian Distribution
    private static double gaussian(double mean, double sigma, double x) {
            double alpha =  1/(sigma*Math.sqrt(2*Math.PI));
            double beta = 0.5*Math.pow(((x-mean)/sigma),2);
            return alpha*Math.exp(-beta);
    }
}
```

B.2 Vector.class

```
public class Vector {
    public double x;
    public double y;
    public double z;

    //Constructor
    public Vector(){
        x = 0;
        y = 0;
        z = 0;
    }
     //Constructor
    public Vector(double xi, double yi, double zi){
        x = xi;
        y = yi;
        z = zi;
    }
```

```
    public double Magnitude() {
        return Math.sqrt(x*x + y*y + z*z);
    }

    public void Normalize(){
        final double tolerance = 0.0001;
        double norm = Math.sqrt(x*x + y*y + z*z);
        if(norm <= tolerance) norm = 1;
        x /= norm;
        y /= norm;
        z /= norm;
        if (Math.abs(x) < tolerance) x = 0;
        if (Math.abs(y) < tolerance) y = 0;
        if (Math.abs(z) < tolerance) z = 0;
    }
    public static Vector vMultiply(Vector u, double v){
        return new Vector(u.x*v, u.y*v, u.z*v);
    }
    public static Vector vDivide(Vector u, double v){
        return new Vector(u.x/v, u.y/v, u.z/v);
    }
    public static Vector vPlus(Vector u, Vector v){
        return new Vector(u.x+v.x, u.y+v.y, u.z+v.z);
    }
    public static Vector vMinus(Vector u, Vector v){
        return new Vector(u.x-v.x, u.y-v.y, u.z-v.z);
    }
    public static double vDot(Vector u, Vector v){
        return (u.x*v.x + u.y*v.y + u.z*v.z);
    }
}
```

Appendix C

JLab - GUI Code

C.1 HTMLViewer.class

```
public class HTMLViewer extends JPanel {
        String stringURL, HTMLtitle;
        JEditorPane html;

    public HTMLViewer(String getTitle, String getURL) {
        this.stringURL = getURL;
        this.HTMLtitle = getTitle;
        getHTMLViewer();
    }

    private void getHTMLViewer(){
        setBorder(BorderFactory.createCompoundBorder(
                    new TitledBorder(new EtchedBorder(), this.HTMLtitle),
                    new BevelBorder(BevelBorder.LOWERED)));
        setLayout(new BorderLayout());
        JScrollPane scrollPane = createHTMLPanel();
        add(scrollPane, BorderLayout.CENTER);
```

```
      }

  private JScrollPane createHTMLPanel(){
  JScrollPane scrollPane = new JScrollPane();
  try {
          URL url = null;
    String path = null;
    try {
path = this.stringURL;
url = getClass().getResource(path);
           } catch (Exception e) {
System.err.println("Failed to open " + path);
url = null;
           }
           if(url != null) {
               html = new JEditorPane(url);
               html.setEditable(false);
               html.addHyperlinkListener(createHyperLinkListener());
JViewport vp = scrollPane.getViewport();
vp.add(html);
           }
       } catch (MalformedURLException e) {
           System.out.println("Malformed URL: " + e);
       } catch (IOException e) {
           System.out.println("IOException: " + e);
       }
  return scrollPane;
  }

  public  HyperlinkListener createHyperLinkListener() {
return new HyperlinkListener() {
    public void hyperlinkUpdate(HyperlinkEvent e) {
```

```
  if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
      if (e instanceof HTMLFrameHyperlinkEvent) {
  ((HTMLDocument)html.getDocument()).processHTMLFrameHyperlinkEvent(
      (HTMLFrameHyperlinkEvent)e);
      } else {
  try {
      html.setPage(e.getURL());
  } catch (IOException ioe) {
      System.out.println("IOE: " + ioe);
  }
      }
  }
      }
  };
     }
}
```

C.2 DecayLabGUI.class

```
public class DecayLabGUI extends JPanel {
        //Main Panel
         static JPanel labPanel;

        //Decay GUI defaults
         static double hf = 3.82;
         static int total_time = 5;
         static int time_intervals = 1;
         static int sample_location = 1;

        //Decay Source Type
         static String radon = "Rn-222";  //p.620 hf=3.823d
         static String radium = "Ra-223";  //p.621 hf=11.43d
```

```
     static String thorium = "Th-227";  //p.621 hf=18.72d
     static String currium = "Cm-240";  //p.621 hf=27d
     static String unknown = "Unknown";  //Randomly Selected Source

    //Decay JComboBox List
     static JComboBox sourceItem = new JComboBox();
     static JComboBox totalTime = new JComboBox();
     static JComboBox timeInt = new JComboBox();
     static JComboBox sampleLocation = new JComboBox();

     //Decay JButtons
     static JButton run;

private static void  buildConstraints(
GridBagConstraints gbc, int gx, int gy, int gw, int gh, int wx, int wy){
        gbc.gridx = gx;
        gbc.gridy = gy;
        gbc.gridwidth = gw;
        gbc.gridheight = gh;
        gbc.weightx = wx;
        gbc.weighty = wy;
}

public static JPanel createDecayLabPanel() {
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Decay Lab Set-Up"),
                new BevelBorder(BevelBorder.LOWERED)));
panel.setLayout(new BorderLayout());
JPanel innerPanel = createDecayOptionPanel();
panel.add(innerPanel, BorderLayout.CENTER);
JPanel runSimulationPanel = createRunSimulationPanel();
```

```
panel.add(runSimulationPanel, BorderLayout.SOUTH);
return panel;
}

private static JPanel createRunSimulationPanel() {
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Run Simulation"),
                new BevelBorder(BevelBorder.LOWERED)));
run = new JButton("Run Experiment");
panel.add(run);
return panel;
}

private static JPanel createDecayOptionPanel() {
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Simulation Option"),
                new BevelBorder(BevelBorder.LOWERED)));
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    pane.setLayout(gridbag);

    //Source Type Label
    buildConstraints(constraints, 0, 0, 1, 1, 30, 30);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.WEST;
    JLabel sourceType = new JLabel("Source Type: ", JLabel.LEFT);
    gridbag.setConstraints(sourceType, constraints);
    pane.add(sourceType);

     //Source Type Option List
```

```
buildConstraints(constraints, 1, 0, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
sourceItem.addItem(radon);  //p.620 hf=3.823d
sourceItem.addItem(radium);  //p.621 hf=11.43d
sourceItem.addItem(thorium);  //p.621 hf=18.72d
sourceItem.addItem(currium);  //p.621 hf=27d
sourceItem.addItem(unknown);
gridbag.setConstraints(sourceItem, constraints);
pane.add(sourceItem);

//Total Time Label
buildConstraints(constraints, 0, 1, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel time = new JLabel("Total Time [days]: ", JLabel.LEFT);
gridbag.setConstraints(time, constraints);
pane.add(time);

 //Total Time Option List
buildConstraints(constraints, 1, 1, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
totalTime.addItem("5");
totalTime.addItem("10");
totalTime.addItem("20");
totalTime.addItem("40");
totalTime.addItem("80");
gridbag.setConstraints(totalTime, constraints);
pane.add(totalTime);

// Time Interval Label
```

```
buildConstraints(constraints, 0, 2, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel ttime = new JLabel("Time Interval [days]: ", JLabel.LEFT);
gridbag.setConstraints(ttime, constraints);
pane.add(ttime);

 //Time Interval Option List
buildConstraints(constraints, 1, 2, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
//timeInt.addItem("0.5");
timeInt.addItem("1");
timeInt.addItem("2");
timeInt.addItem("5");
gridbag.setConstraints(timeInt, constraints);
pane.add(timeInt);

//Sample Location Label
buildConstraints(constraints, 0, 3, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel location = new JLabel("Sample Location: ", JLabel.LEFT);
gridbag.setConstraints(location, constraints);
pane.add(location);

//Sample Location Option List
buildConstraints(constraints, 1, 3, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
sampleLocation.addItem("1");
sampleLocation.addItem("2");
```

```
        sampleLocation.addItem("3");
        sampleLocation.addItem("4");
        gridbag.setConstraints(sampleLocation, constraints);
        pane.add(sampleLocation);
        constraints.fill = GridBagConstraints.BOTH;
    return pane;
    }
}
```

C.3 SternGerlachLabGUI.class

```
public class SternGerlachLabGUI extends JPanel {
        //Main Panel
        static JPanel labPanel;

        //Atom Type
        public static String potassium = "Potassium";
        public static String silver = "Silver";

        //Stern-Gerlach JComboBox List
        public static JComboBox sourceItem = new JComboBox();
        public static JComboBox temperature = new JComboBox();
        public static JComboBox magneticField = new JComboBox();
        public static JComboBox detectorStartLocation = new JComboBox();
        public static JComboBox incrementSize = new JComboBox();
        public static JComboBox numberOfIncrement = new JComboBox();
        public static JComboBox numberOfEvent = new JComboBox();

        //Stern-Gerlach JButtons
        public static JButton run;

    private static void  buildConstraints(
```

```
GridBagConstraints gbc, int gx, int gy,
int gw, int gh, int wx, int wy){
        gbc.gridx = gx;
        gbc.gridy = gy;
        gbc.gridwidth = gw;
        gbc.gridheight = gh;
        gbc.weightx = wx;
        gbc.weighty = wy;
}

public static JPanel createSternGerlachLabPanel() {
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createCompoundBorder(
        new TitledBorder(new EtchedBorder(), "Stern-Gerlach Lab Set-Up"),
        new BevelBorder(BevelBorder.LOWERED)));
panel.setLayout(new BorderLayout());
JPanel innerPanel = createSternGerlachOptionPanel();
panel.add(innerPanel, BorderLayout.CENTER);
JPanel runSimulationPanel = createRunSimulationPanel();
panel.add(runSimulationPanel, BorderLayout.SOUTH);
return panel;
}

private static JPanel createRunSimulationPanel() {
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Run Simulation"),
                new BevelBorder(BevelBorder.LOWERED)));
run = new JButton("Run Experiment");
panel.add(run);
return panel;
}
```

```
private static JPanel createSternGerlachOptionPanel() {
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createCompoundBorder(
                new TitledBorder(new EtchedBorder(), "Simulation Option"),
                new BevelBorder(BevelBorder.LOWERED)));
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    pane.setLayout(gridbag);

    //Source Type Label
    buildConstraints(constraints, 0, 0, 1, 1, 30, 30);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.WEST;
    JLabel sourceType = new JLabel("Source Type: ", JLabel.LEFT);
    gridbag.setConstraints(sourceType, constraints);
    pane.add(sourceType);

     //Source Type Option List
    buildConstraints(constraints, 1, 0, 1, 1, 30, 30);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.EAST;
    sourceItem.addItem(potassium);  // a_mass = 39.1
    sourceItem.addItem(silver);  // a_mass = 108
    gridbag.setConstraints(sourceItem, constraints);
    pane.add(sourceItem);

    //Temperature Label
    buildConstraints(constraints, 0, 1, 1, 1, 30, 30);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.WEST;
    JLabel temp = new JLabel("Temperature [K]: ", JLabel.LEFT);
```

```
gridbag.setConstraints(temp, constraints);
pane.add(temp);

 //Temperature Option List
buildConstraints(constraints, 1, 1, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
temperature.addItem("388.15");
//temperature.addItem("1000");
gridbag.setConstraints(temperature, constraints);
pane.add(temperature);

//Magnetic Field Label
buildConstraints(constraints, 0, 2, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel bField = new JLabel(
"Magnetic Field Gradient [Tesla/m]: ", JLabel.LEFT);
gridbag.setConstraints(bField, constraints);
pane.add(bField);

 //Magnetic Field Option List
buildConstraints(constraints, 1, 2, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
magneticField.addItem("0");
magneticField.addItem("50");
magneticField.addItem("100");
magneticField.addItem("150");
gridbag.setConstraints(magneticField, constraints);
pane.add(magneticField);
```

```
//Detector Starting Location Label
buildConstraints(constraints, 0, 3, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel detectorLocation = new JLabel(
"Detector Starting Location: [m] ", JLabel.LEFT);
gridbag.setConstraints(detectorLocation, constraints);
pane.add(detectorLocation);

//Sample Location Option List
buildConstraints(constraints, 1, 3, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
detectorStartLocation.addItem("-0.005");
detectorStartLocation.addItem("-0.001");
detectorStartLocation.addItem("0.000");
detectorStartLocation.addItem("0.001");
detectorStartLocation.addItem("0.005");
gridbag.setConstraints(detectorStartLocation, constraints);
pane.add(detectorStartLocation);
constraints.fill = GridBagConstraints.BOTH;

//Detector Increment Label
buildConstraints(constraints, 0, 4, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel detectorInc = new JLabel(
"Increment Size: [m] ", JLabel.LEFT);
gridbag.setConstraints(detectorInc, constraints);
pane.add(detectorInc);

//Detector Increments Option List
```

```
buildConstraints(constraints, 1, 4, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
incrementSize.addItem("0.0002");
incrementSize.addItem("0.0004");
incrementSize.addItem("0.0006");
gridbag.setConstraints(incrementSize, constraints);
pane.add(incrementSize);
constraints.fill = GridBagConstraints.BOTH;

//Number of Increments Label
buildConstraints(constraints, 0, 5, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.WEST;
JLabel numberInc = new JLabel(
"Number of Increments: ", JLabel.LEFT);
gridbag.setConstraints(numberInc, constraints);
pane.add(numberInc);

//Number of Increments Option List
buildConstraints(constraints, 1, 5, 1, 1, 30, 30);
constraints.fill = GridBagConstraints.NONE;
constraints.anchor = GridBagConstraints.EAST;
numberOfIncrement.addItem("1");
numberOfIncrement.addItem("5");
numberOfIncrement.addItem("10");
numberOfIncrement.addItem("50");
gridbag.setConstraints(numberOfIncrement, constraints);
pane.add(numberOfIncrement);
constraints.fill = GridBagConstraints.BOTH;

//Number of Events Label
```

```
        buildConstraints(constraints, 0, 6, 1, 1, 30, 30);
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.WEST;
        JLabel numberEvent = new JLabel(
        "Number of Atoms Generated: ", JLabel.LEFT);
        gridbag.setConstraints(numberEvent, constraints);
        pane.add(numberEvent);

        //Number of Increments Option List
        buildConstraints(constraints, 1, 6, 1, 1, 30, 30);
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.EAST;
        numberOfEvent.addItem("1000");
        numberOfEvent.addItem("5000");
        numberOfEvent.addItem("10000");
        numberOfEvent.addItem("50000");
        gridbag.setConstraints(numberOfEvent, constraints);
        pane.add(numberOfEvent);
        constraints.fill = GridBagConstraints.BOTH;
    return pane;
    }
}
```

Appendix D

JLab - JApplet Code

D.1 NuclearDecayApplet.class

```
public class NuclearDecayApplet extends JApplet
    implements ActionListener, ItemListener {
    int x_size = 800;
    int y_size = 600;
    JPanel topPanel;
    JPanel theoryPanel;
    JPanel decayTools;
    JEditorPane html;

    //Simulation Input
    double hf = 3.82;  //Default decay source
    int total_time = 5;  //Default total time
    int sample_location = 1; //Default location
    int time_intervals = 1; //Default time intervals

    public void init() {
```

```
//Set-up Main Panel
setSize(x_size,y_size);
topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

//Set-up Panels
decayTools = DecayLabGUI.createDecayLabPanel();
topPanel.add(decayTools, BorderLayout.WEST);

HTMLViewer theoryPanel = new HTMLViewer("Theory", "/DecayInfo.html");
topPanel.add(theoryPanel, BorderLayout.CENTER);

//Application ActionListener List
DecayLabGUI.sourceItem.addItemListener(this);
DecayLabGUI.totalTime.addItemListener(this);
DecayLabGUI.timeInt.addItemListener(this);
DecayLabGUI.sampleLocation.addItemListener(this);
DecayLabGUI.run.addActionListener(this);
}

public void itemStateChanged(ItemEvent ievent) {
Object isource = ievent.getSource();

if (isource == DecayLabGUI.sourceItem) {
    Object newSourceItem = ievent.getItem();
    if (DecayLabGUI.radon.equals(newSourceItem) == true) {
        hf = 3.82;
    }
    else if (DecayLabGUI.radium.equals(newSourceItem) == true){
        hf = 11.43;
    }
```

```
    else if (DecayLabGUI.thorium.equals(newSourceItem) == true){
        hf = 18.72;
    }
    else if (DecayLabGUI.currium.equals(newSourceItem) == true){
        hf = 27.00;
    }
    else {
        double randomSource = 3*Math.random();
        if (randomSource <= 1){
            hf = 5.01;  //Bi-210 p.619
        } else if (randomSource > 1 && randomSource <= 2){
            hf = 46.6; // Hg-203 p.618
        } else {
            hf = 10.0; //Ac-225 p.621
        }
    }
}
else if (isource == DecayLabGUI.totalTime) {
    Object newTotalTime = ievent.getItem();
    String s_total_time = newTotalTime.toString();
    total_time = Integer.parseInt(s_total_time);

}
else if (isource == DecayLabGUI.timeInt) {
    Object newTimeInt = ievent.getItem();
    String s_time_intervals = newTimeInt.toString();
    time_intervals = Integer.parseInt(s_time_intervals);
}
else if (isource == DecayLabGUI.sampleLocation) {
    Object newSampleLocation = ievent.getItem();
    String s_sample_location = newSampleLocation.toString();
    sample_location = Integer.parseInt(s_sample_location);
```

```
    }
    repaint();
    }

    public void actionPerformed(ActionEvent event) {
        Object source = event.getSource();
        if (source == DecayLabGUI.run) {

        //Re-set number of atoms & decay
        int bins = total_time/time_intervals;
        double yArray[] =
        DecayProcess.decayProcess(
        hf, total_time, time_intervals, sample_location);
        double xArray[] =
        DecayProcess.binProcess(total_time, time_intervals);
        Plot2D.linearFit = false;
        Plot2D dataPlot = new Plot2D(xArray, yArray);
        dataPlot.getTitle("N(t) vs. Time");
        dataPlot.getXLabel("Time [day]");
        dataPlot.getYLabel("N(t)");
        dataPlot.setVisible(true);
        }
    }
}
```

D.2 SternGerlachApplet.class

```
public class SternGerlachApplet
    extends JApplet implements ActionListener, ItemListener {
    int x_size = 800;
    int y_size = 600;
    JPanel topPanel;
```

```
JPanel theoryPanel;
JPanel sgTools;
JEditorPane html;

//Simulation Default Input
static double atomic_mass = 39.1;  //Atomic Mass
static double temperature = 388.15;  //Temperature in oven [k]
static double magneticField = 0;  //Magnetic Field [Tesla/m]
static double detectorStartLocation = -0.005; //Detectors starting location
static double incrementSize = 0.0002;  //Detector increment on the y-axis
static int numberOfIncrements = 1;
static int numberOfEvent = 1000;  //Number of atoms that will be generated

public void init() {
//Set-up Main Panel
setSize(x_size,y_size);
topPanel = new JPanel();
topPanel.setLayout(new BorderLayout());
getContentPane().add(topPanel);

//Set-up Panels
sgTools = SternGerlachLabGUI.createSternGerlachLabPanel();
topPanel.add(sgTools, BorderLayout.WEST);

HTMLViewer theoryPanel = new HTMLViewer("Theory", "/sgInfo.html");
topPanel.add(theoryPanel, BorderLayout.CENTER);

//Application ActionListener List
SternGerlachLabGUI.sourceItem.addItemListener(this);
SternGerlachLabGUI.temperature.addItemListener(this);
SternGerlachLabGUI.magneticField.addItemListener(this);
SternGerlachLabGUI.detectorStartLocation.addItemListener(this);
```

```
SternGerlachLabGUI.incrementSize.addItemListener(this);
SternGerlachLabGUI.numberOfIncrement.addItemListener(this);
SternGerlachLabGUI.numberOfEvent.addItemListener(this);
SternGerlachLabGUI.run.addActionListener(this);
}

public void itemStateChanged(ItemEvent ievent) {
Object isource = ievent.getSource();
if (isource == SternGerlachLabGUI.sourceItem) {
    Object newSourceItem = ievent.getItem();
    if (SternGerlachLabGUI.potassium.equals(newSourceItem) == true) {
        atomic_mass = 39.1;
    }
    else if (SternGerlachLabGUI.silver.equals(newSourceItem) == true){
         atomic_mass = 69.1;
    }
}
else if (isource == SternGerlachLabGUI.temperature) {
    Object newTemperature = ievent.getItem();
    String sTemperature = newTemperature.toString();
    temperature = Double.parseDouble(sTemperature);
    System.out.print("Working");
}
else if (isource == SternGerlachLabGUI.magneticField) {
    Object newMagneticField = ievent.getItem();
    String s_magneticField = newMagneticField.toString();
    magneticField = Double.parseDouble(s_magneticField);
    //need to convert to force
}
else if (isource == SternGerlachLabGUI.detectorStartLocation) {
    Object newDetectorStartLocation = ievent.getItem();
    String s_detectorLocation = newDetectorStartLocation.toString();
```

```
        detectorStartLocation = Double.parseDouble(s_detectorLocation);
    }
    else if (isource == SternGerlachLabGUI.incrementSize) {
        Object newIncrementSize = ievent.getItem();
        String s_incrementSize = newIncrementSize.toString();
        incrementSize = Double.parseDouble(s_incrementSize);
    }
    else if (isource == SternGerlachLabGUI.numberOfIncrement) {
        Object newNumberOfIncrement = ievent.getItem();
        String s_numberOfIncrement = newNumberOfIncrement.toString();
        numberOfIncrements = Integer.parseInt(s_numberOfIncrement);
    }
    else if (isource == SternGerlachLabGUI.numberOfEvent) {
        Object newNumberOfEvent = ievent.getItem();
        String s_numberOfEvent = newNumberOfEvent.toString();
        numberOfEvent = Integer.parseInt(s_numberOfEvent);
    }
    repaint();
    }
    public void actionPerformed(ActionEvent event) {
        Object source = event.getSource();
        if (source == SternGerlachLabGUI.run) {
            int count = 0;
            int countArray[] = new int[numberOfIncrements];
            //Initilizes the Stern-Gerlach simulation
            SternGerlachSimulation runSim = new SternGerlachSimulation(
            numberOfEvent, atomic_mass, temperature, magneticField,
            detectorStartLocation, incrementSize, numberOfIncrements);
        }
    }
}
```

Vita

Malachi Schram was born on July 16, 1974 in Ottawa, Ontario, Canada. He graduated from Louis-Riel High School in Ottawa, Ontario in 1992. He then attended South Georgia College for part of his undergraduate education. Malachi not only studied hard but was on the collegiate baseball team and helped as a peer counselor/tutor. After two years, he transfered to Valdosta State University to finish his BS in Physics. During his stay at Valdosta State University he was inducted into the Physics, Mathematics, and Co-Operative Education Honor Society. In 1998, Malachi was awarded the years Outstanding Graduating Student in Physics. The following year, he received his BA in Mathematics. He was awarded a NSERC Undergraduate Student Research Award during the summer of 1999 to work with Dr. Oakham as part of the Canadian ATLAS group in Ottawa, building a Calorimeter module for the FCAL detector. He moved to Knoxville, TN in the fall of 1999 to pursue his MS in Physics at the University of Tennessee. He was awarded the SARIF Graduate Research Award and worked with Dr. Kamyskov on several projects. He later joined Dr. Breining to work on developing online physics laboratories. Presently, he is finishing his MS in Physics and is expected to graduate in the fall of 2002.