A Simulation of B Meson Events in the LHCb and the Effects of Different Resolution Detectors

James Peters
(201122706)
Department of Physics
University of Liverpool

Abstract

In this report the effects of resolution on the accuracy of the LHCb are investigated. The parameters tested include the average decay time from 1421 events, and the average gaussian distance on selected events. The most efficient resolution calculated for the detector was between $10^{-6}m$ and $10^{-5}m$. Two algorithms are included that are used to fit a straight line to set of 3 dimensional points in terms of a direction and position vector, and then finding the closest point to the set of lines produced by the B meson decay products. The algorithms implementation is then tested with a 3D graph to verify its ability to fit straight lines, and finally the benefits of multithreaded computing is shown.

I. Introduction

For our project we were set the task of simulating B meson production and detection with the LHCb detector. We were provided with a large data-set that included thousands of simulations of B0 particles – which decayed into 2 Muons and 2 Pions. This included the true decay point of the B0 meson and its momentum and then the momentums of all the decay products. Therefore, using the true decay point and the decay product momentum's we could propagate the particles through our simulated detector which would produce a set of coordinates at each point the particle hit a slice of the detector.

Using this data a straight line can be fit in 3 dimensions and then all the lines can be extrapolated towards a common origin.

My main contribution to this task was researching the algorithms used to fit a 3 dimensional data-set to a straight line and then finding the common origin they all share, and then writing an abstract class as a framework for all the separate simulations.

II. Description of Code

Initially I thought this would be a simple task to fit a straight line to a set of points, however in 3 dimensions this becomes slightly more difficult. A lot of algorithms I found were for fitting a 3D plane to the points, when in fact I need a position and direction vector that best fit the point.

I found that Principal Component Analysis was needed and looked up an algorithm to implement. This was as follows:

Algorithm 1 *Let X be a $n \times 3$ matrix. Where n is the number of points, and each row represents the x,y,z of that point.*

$$X = \begin{bmatrix} x_i & y_i & z_i \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix}$$

To find the position vector for the line you simply calculate the average of all the points:

$$(\bar{x}, \bar{y}, \bar{z}) = \frac{1}{n}\sum_{i=1}^{n}(x_i, y_i, z_i)$$

Let PX be the linear transformation $(x_i, y_i, z_i) - (\bar{x}, \bar{y}, \bar{z})$

And the 3×3 matrix $M = (PX)^T PX$
M can be rearranged to $X^T PX$

The eigenvector $\vec{e}$ with the largest eigenvalue is the direction vector of the line. Giving a final equation for the line of best fit:

$$\{(\bar{x}, \bar{y}, \bar{z}) + t\vec{e} \ : \ t \in \mathbb{R}\}$$

[1]

Now it was possible to fit 4 lines to the 4 different decay particles the goal was now to find the point of intersection of all 4 of the lines. Since they all will not meet at exactly the same point, a Least Squares approach is needed to minimise the sum of the perpendicular distances from a unique solution.

To solve this I found a paper online that provided an algorithm that solves for this point. Below is the steps of the algorithm without derivations:

Algorithm 2 *Let n_j be the direction vector for line j*
Let a_j be the position vector for line j

So the line is of the form $p = a + tn$

Let a linear system of equations be:

$$Rp = q$$

$$R = \sum_{j=1}^{K}(I - n_j n_j^T) \quad , \quad q = \sum_{j=1}^{K}(I - n_j n_j^T)a_j$$

Where I is a 3 x 3 Identity matrix.
The system can then be solved for p using a Gaussian Elimination algorithm, also known as reduced row echelon. [2]

i. StraightLineFactory.java

This class takes an array of vectors and fits a straight line to them. To perform matrix multiplications and other matrix manipulations the Apache Commons library was used. [3]

So for example a Vector or Matrix could be initialised as follows:

```
//Creates a vector (0,0,0)
double pos = new double[] {0,0,0};
RealVector v = new ArrayRealVector(pos);

//Creates an empty 3x3 Matrix
RealMatrix M = new BlockRealMatrix(3,3);
```

Algorithm 1 was then implemented using these methods:

```
//Adds one vector to another
v1 = v1.add(v2);

//Divides a vector by a scalar value n
v.mapDivideToSelf(n);

//Inserts a vector (v) as a row (n) of
    a Matrix (M)
M.setRowVector(n,v);

//Calculate the multiplication of a
    Matrix to its transpose
RealMatrix MTM =
    M.transpose().multiply(M);

//Calculate the Eigen vectors and value
    of a Matrix
EigenDecomposition ed = new
    EigenDecomposition(XPX);
```

This was all that was needed to calculate the position and direction vectors using Algorithm 1.

ii. FindNearestPoint.java

This class takes arrays of direction and position vectors and uses Algorithm 2 to find the nearest point to each line.

The Apache commons library was again used to implement this algorithm using the following function:

```
//Create the Matrix R, Identity Matrix
    I and Vector q
I = MatrixUtils.
        createRealIdentityMatrix(dim);
R = MatrixUtils.
        createRealMatrix(dim, dim);
q = new ArrayRealVector(dim);

//Sum of R and q according to algorithm
for(int i = 0; i < n; i++) {
   R = R.add(
        I.subtract(
                d[i].outerProduct(d[i])
        ));

   q = q.add(
        I.subtract(d[i].outerProduct(d[i]))
                .operate(a[i]));
}
```

Where d is the array of direction vectors and a the array of position vectors. This then meant that we had the equation $Rp = q$ and needed to solve for p. This was achieved using a Gaussian Elimination algorithm.

iii. GaussianElimination.java

This class performed Gaussian elimination on a linear system, also known as reduced row echelon. I found a class online that would take a Matrix in the form of a double array and then a vector as a single array, and then perform row echelon to return the solution p. [4]

The notation for using the class was then:

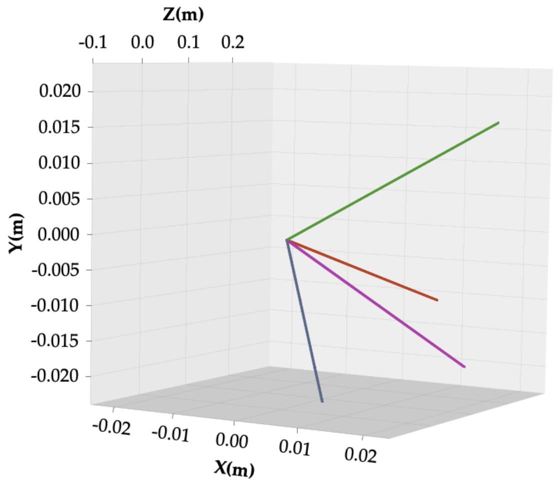
```
GaussianElimination g =
new GaussianElimination(
        R.getData(),    //Matrix of R
        q.toArray());   //Vector q

RealVector solution = g.getSolution();
```

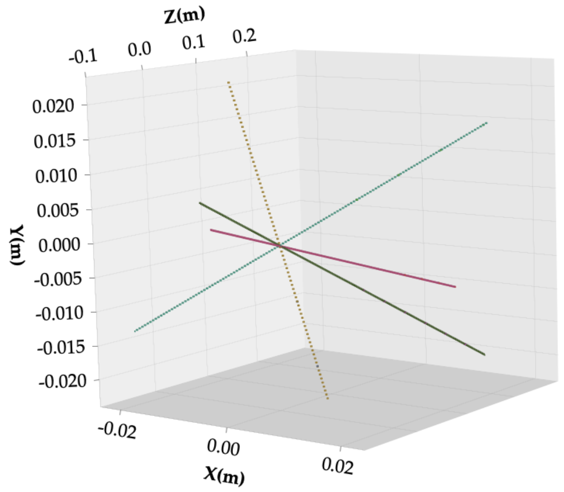
iv. Display3DEvent.java

To verify the previous classes worked a 3D Graphing library was used to display out data. The library used was Orson Charts. [5]

Using all the previous classes, and classes contributed by other members of the team to import the dataset and perform the simulation it was then possible to plot the true path the particles took through the detector, and then also plot the straight lines calculated from the detectors. So for example, event 113 followed this true path:



And then using the detection points and fitting straight lines to the data using the classes defined above creates the following graph:



So the program worked and could fit straight lines to data created by the simulation, now it was time to change variables of the simulation to see their effects.

v. Simulation.java

I helped create an abstract class that would be extended by each type of simulation we wanted. The base Simulation class had methods used to start a loop of the simulation and store the data produced into a Histogram. And then helper methods such as saving the histogram to a CSV file.

The abstract methods to be implemented by child classes were as follows:

```
/**
 * Called after every Loop
 *
 * @param event
 * @return EventSimulation
 * @throws Exception
 */
public abstract EventSimulation
    EventLoop(int eventId, int
    currentLoop) throws Exception;


/**
 * Called after each simulation has
    finished.
 * @param currentLoop
 * @throws Exception
 */
public abstract void postSimulation(int
    currentLoop) throws Exception;

/**
 *  Called after all simulation loops
    have completed.
 */
public abstract void
    postSimulationLoop();


/**
 * @return GraphValues object that
    contains all
 * the information to be plotted onto
    the final graph.
 */
public abstract GraphValues
    configureGraph();
```

So the simulation class would run the EventSimulation class for every event and loop that process a certain amount of times. So then for each loop the child class could be programmed to perform a different action, such as change the smearing, detector thickness, material or spacing.

I discovered that when running this for around 1500 events and looping that 10 times it took a long time to simulate. This was due to the single threaded design so far, so only one core from the CPU was being utilised by the program.

To overcome this the ExecutorService class was used to queue all the events and then run them all in parallel on all cores and threads of the CPU. This greatly sped up the running time of the program, especially when running the program for long periods multithreading improved performance by over 250%. The improved performance is shown below:

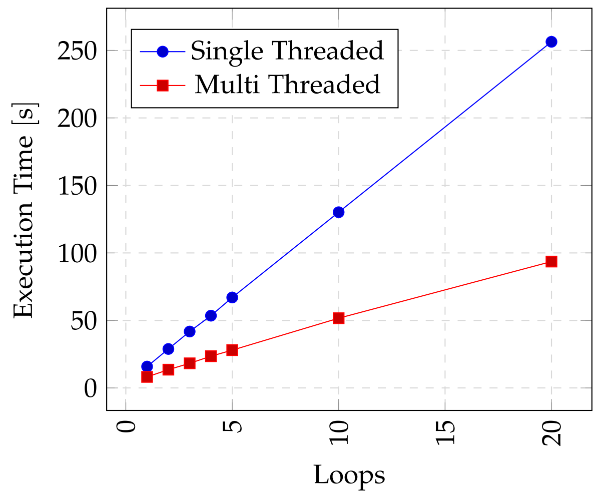


Figure 1: *Graph comparing Single and Multithreaded execution times*

The tests were run on a 6 core 12 thread CPU a reason why the increase in performance is more drastic compared to just a 4 core processor.

The ExecutorService class was used as follows:

```
//Creates ExecutorService service.
ExecutorService service =
    Executors.newCachedThreadPool();
```

And then tasks can be added to the queue as follows:

```
Future<EventSimulation> f =
    service.submit(() -> {

   EventSimulation sim = new
       EventSimulation(...);
   sim.start();
}
```

When submitted a task returns a Future object that can be used to check if the given task has finished. So all the Future were added to an ArrayList and then a while loop was used to ensure all the tasks had finished before moving on.

The while loop worked by calling a boolean function that would loop over ever Future and check whether it had finished, if not the function would immediately return false. The loop would keep checking until all tasks returned true.

The next library implemented was Michael Thomas Flanagan's Java Scientific Library. This library allows you to fit various curves to data in 2 dimensions. So in our program we fit exponential curves and gaussian distributions. The exponential curve was fit to the distribution of decay times in the data-set to find the average decay time. And gaussian curves were fit to data such as the average point away from the true decay vertex. [6]

A regression was created using the following code:

```
//Setup regression in x and y with
    error bars on y.
Regression reg = new
    Regression(x,y,errorY);

//Call your regression type
//e.g exponential, gaussian.
reg.exponentialSimple();

//Best estimates of Mean and its error.
mean = -reg.getBestEstimates()[0];
error = reg.getBestEstimatesErrors()[0];
```

This completed the main Simulation class. Another class was extended from the Simulation class that would calculate the average distance from the true vertex instead of decay time.

III. Effects of Smearing

To investigate the effects of smearing I created two classes that extended Simulation.java and AveragePointSimulation.java. These would calculate the average decay time and the average distance from the true vertex respectively.

The first class DetectorSmearingSimDecayTimes.java had the following variables:

```
//Starting smear - lowest resolution.
double startSmear = 0.0001;
//Smear step, smear decreased
    logarithmically by 10^-stepSmear.
double stepSmear = 0.05;
//Number of different smears to
    calculate from startSmear.
int numberOfSmears = 100;
//Number of times to run the
    simulation, each time calculating a
    more accurate result and error.
int accuracy = 10;
```

These values could be changed to see the effects at different resolutions.

The accuracy value was implemented by running the simulation for all 1500 events multiple times, each time averaging the calculated decay time and finding the standard error. This meant that extremely precise values could be achieved by allowing the simulation to run longer.

This class used the postSimulation call from the Simulation.java class. The simulation was ran for all events with 0 smearing, producing a baseline data-set. This data-set could then have smearing applied by simply adding random gaussian numbers with a standard deviation of the resolution. This meant that the whole simulation didn't need to be run for each smearing value, which saves a lot of time. And also has the advantage of producing accurate results.

The differences between an accuracy of 1 and an accuracy of 10 is shown in the following graph:

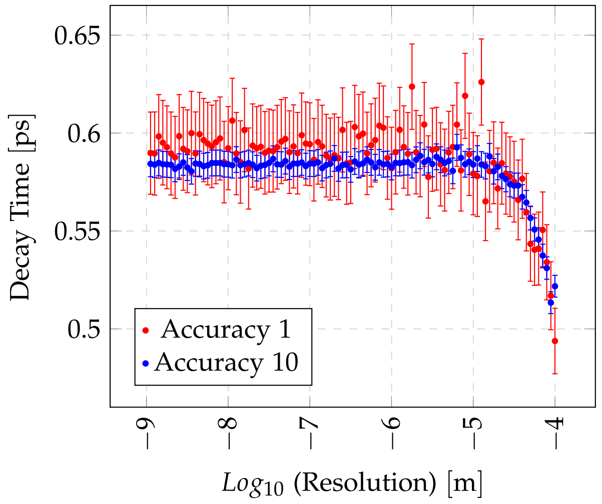


Figure 2: *Graph comparing different accuracies of simulation*

This graph clearly shows the benefits of running the simulation longer for higher accuracy. The error bars become a lot smaller and thus the dataset more accurate.

If the graph is now just plotted with an accuracy of 10 the effects of smearing can be explored, as shown in figure 3.

It's clear that after a certain value the benefit of having a higher resolution detector becomes pointless. Between $Log_{10}(r)$=-6 and $Log_{10}(r)$=-5 is the range between which the lowest resolution performs best.

This is an important reason why a simulation must be run before building such a project. There is no need to spend additional time and money to get a higher resolution detector if it doesn't provide any benefit to the system.

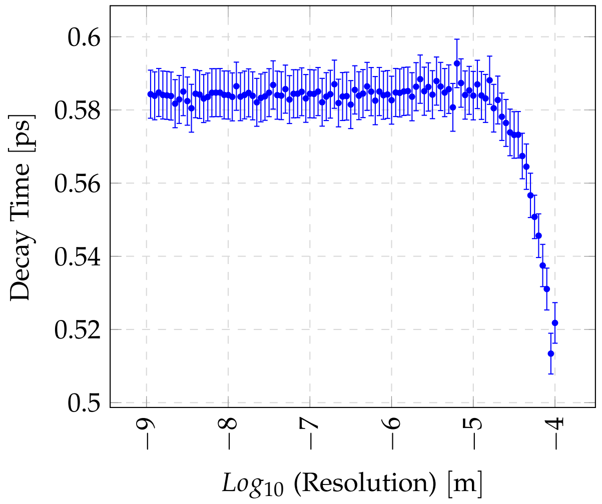


Figure 3: *Graph of Calculated Decay Time against Log_{10} (Resolution)*

To confirm my findings I also ran the simulation comparing the resolution to the average distance from the true vertex given in the reference CSV file. This should provide the same or similar results as the decay time is calculated using distance as well.

This was achieved with the DetectorSmearingSimAveragePoint class. This class extended the AveragePointSimulation class and looped over the same event multiple times and calculated a range of smearing values.

This simulation had to be performed with just one event because the point around the true vertex followed a gaussian distribution. So therefore if the average distance was ran for 1500 different events, we would just be left with 1500 random gaussian numbers unrelated to each other. To achieve a good accuracy instead the simulation was ran on one event 1500 times. The accuracy could be improved even further if this was then run over multiple events, however this would require a significant amount of time/processing power. For example, after multithreading this simulation would take approximately 12 hours to run. Since we would be repeating the simulation of each event 1000 times, and then with 1500 events that's 1.5 million events.

This simulation produced the graph shown in Figure 4 where the average distance is the gaussian error on the measurement.

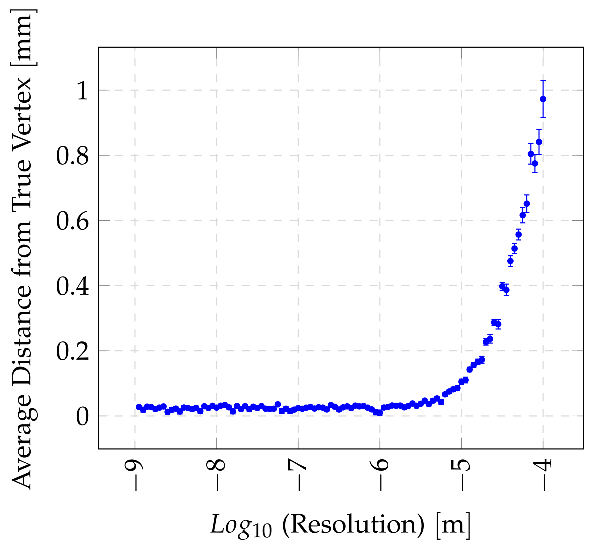


Figure 4: *Graph of Average Distance from True Vertex against Log_{10} (Resolution)*

This simulation agrees with the previous showing that you achieve negligible benefits from a higher resolution below 10^{-6}m. The accuracy of the detector becomes so good that most of the random noise is due to other effects, such as the material of the detector and the scattering it causes. This is where thinner and different types of materials would be more beneficial - the physicality of achieving such goals however is limited, an ideal detector would obviously be as thin as possible with a super high resolution and cause almost no scattering. A simulation therefore can help in deciding which components are least beneficial and where you can cut costs.

IV. Conclusions

To conclude, this simulation is a good indicator of the maximum resolution needed for accurate results in the LHCb detector - increasing it beyond that would cost more money and require scientific advances to design the materials to perform at that resolution all whilst providing no extra benefit to the detector.

The simulation has also shown how multithreading such a program greatly benefits the execution time when running for long periods. Gains of up to 300% were found, and would only increase linearly the longer the simulation was ran, and the amount of cores and threads a CPU has.

To further improve the workflow of the project next time I would suggest a dependancy manager such a Maven or Gradle, this would enable everybody to quickly share dependancies and keep everyones workspace in sync without requiring time to update them each time someone added one.

I would also have changed the way the Simulation class was implemented, since when it was first created it was made with just the decay times in mind, later on the idea of calculating the average distance was added in a new class, this resulted in some messy code which could be cleaned up and made more modular.

References

[1] M. H. (https://math.stackexchange.com/users/11667/michael-hardy), *Best fit line with 3d points*, Mathematics Stack Exchange, URL:https://math.stackexchange.com/q/1612462 (version: 2016-01-14). eprint: `https://math.stackexchange.com/q/1612462`. [Online]. Available: `https://math.stackexchange.com/q/1612462`.

[2] J. Traa, "Least-squares intersection of lines", University of Illinois at Urbana-Champaign, Tech. Rep., 2013.

[3] A. C. D. Team. [Online]. Available: `http://commons.apache.org/`.

[4] M. Bhojasia, *Java program to implement gaussian elimination algorithm*, Dec. 2013. [Online]. Available: `https://www.sanfoundry.com/java-program-gaussian-elimination-algorithm/`.

[5] O. R. Limited, *Orson chart library*. [Online]. Available: `http://www.object-refinery.com/orsoncharts/`.

[6] [Online]. Available: `https://www.ee.ucl.ac.uk/~mflanaga/java/`.

Appendices

All code can be found online at GitHub at: `https://github.com/jameslfc19/PHYS488-Project`

A. StraightLineFactory.java

```
public class StraightLineFactory {

        RealVector mean;
        int dimensionSize = 0;
        EigenDecomposition ed;
        public RealVector[] rawVectors;

        private boolean isValid = true; //Set to false if only one point is provided.

        //Find straight from set of 3D points (Use ArrayRealVector).
        public StraightLineFactory(RealVector[] vectors) throws Exception {

                this.rawVectors = vectors;

                //Take first vectors dimensions as the dimension size for the set.
                if(dimensionSize == 0) dimensionSize = vectors[0].getDimension();

                if(vectors.length <= 1) isValid = false;

                //Create X Matrix
                RealMatrix X = new BlockRealMatrix(vectors.length,3);
                for(int i = 0; i < vectors.length; i++) {
                        X.setRowVector(i, vectors[i]);
                }

                //Calculate the mean of the group of vectors.
                mean = new ArrayRealVector(dimensionSize);
                for(RealVector v : vectors) {
                        if(v.getDimension() != dimensionSize) throw new Exception("Dimension
                            of vector not :"+dimensionSize+" for vector "+v);
                        mean = mean.add(v);
                }
                mean.mapDivideToSelf(vectors.length);

                //Create PX Matrix
                RealMatrix PX = new BlockRealMatrix(vectors.length,3);
                for(int i = 0; i < vectors.length; i++) {
                        PX.setRowVector(i, vectors[i].subtract(mean));
                }

                //Compute the covariance of the vectors as a Matrix
                RealMatrix XPX = X.transpose().multiply(PX);

                //Calculate Eigenvectors and Eigenvalues.
                ed = new EigenDecomposition(XPX);
        }

        public StraightLineFactory(ArrayList<RealVector> vectors) throws Exception {
                this(vectors.toArray(new RealVector[vectors.size()]));
        }

        public RealVector getDirectionVector() {
                return ed.getEigenvector(0);
        }

        public RealVector getOriginVector() {
                return mean;
        }

        public boolean isValid() {
                return isValid;
        }
}
```

B. FindNearestPoint.java

```
public class FindNearestPoint {

        private int dim; //Dimension of system.
        private int n; //Number of lines.
        private RealVector[] a, d; //Arrays of: Position Vectors, Direction Vectors.

        private RealMatrix I;
        private RealMatrix R;
        private RealVector q;

        private RealVector solution;

        public FindNearestPoint(RealVector[] a, RealVector[] d, int dimensions) throws
            Exception {
                if(a.length != d.length) throw new Exception("Number of position vectors
                    doesn't match number of direction vectors.");
                this.n = a.length;
                this.dim = dimensions;
                this.a = a;
                this.d = d;


                findNearestPoint();
        }

        private void findNearestPoint() {
                //Create the Matrix R, Identity Matrix I and Vector q
                I = MatrixUtils.createRealIdentityMatrix(dim);
                R = MatrixUtils.createRealMatrix(dim, dim);
                q = new ArrayRealVector(dim);

                //Performs summation of R and q according to algorithm
                for(int i = 0; i < n; i++) {
                        R = R.add(I.subtract(d[i].outerProduct(d[i])));
                        q = q.add(I.subtract(d[i].outerProduct(d[i])).operate(a[i]));
                }

                GaussianElimination g = new GaussianElimination(R.getData(),q.toArray());
                solution = g.getSolution();
        }

        /**
         * @return RealVector of true position in metres.
         */
        public RealVector getPoint() {
                return solution;
        }


}
```

C. GaussianElimination.java

```
** Java Program to Implement Gaussian Elimination Algorithm
 **/
 public class GaussianElimination
{

        private RealVector solution;

        public GaussianElimination(double[][] A, double[] B) {
                solve(A,B);
        }

    public void solve(double[][] A, double[] B)
    {
        int N = B.length;
        for (int k = 0; k < N; k++)
        {
            /** find pivot row **/
            int max = k;
            for (int i = k + 1; i < N; i++)
                if (Math.abs(A[i][k]) > Math.abs(A[max][k]))
                    max = i;

            /** swap row in A matrix **/
            double[] temp = A[k];
            A[k] = A[max];
            A[max] = temp;

            /** swap corresponding values in constants matrix **/
            double t = B[k];
            B[k] = B[max];
            B[max] = t;

            /** pivot within A and B **/
            for (int i = k + 1; i < N; i++)
            {
                double factor = A[i][k] / A[k][k];
                B[i] -= factor * B[k];
                for (int j = k; j < N; j++)
                    A[i][j] -= factor * A[k][j];
            }
        }

        /** Print row echelon form **/
        //printRowEchelonForm(A, B);

        /** back substitution **/
        double[] solution = new double[N];
        for (int i = N - 1; i >= 0; i--)
        {
            double sum = 0.0;
            for (int j = i + 1; j < N; j++)
                sum += A[i][j] * solution[j];
            solution[i] = (B[i] - sum) / A[i][i];
        }

        this.solution = new ArrayRealVector(solution);
        if(this.solution.isNaN()) this.solution = new ArrayRealVector(0);


        /** Print solution **/
        //printSolution(solution);
    }
    /** function to print in row    echleon form **/
    public void printRowEchelonForm(double[][] A, double[] B)
    {
        int N = B.length;
        System.out.println("\nRow Echelon form : ");
        for (int i = 0; i < N; i++)
           {
               for (int j = 0; j < N; j++)
                   System.out.printf("%.3f ", A[i][j]);
               System.out.printf("| %.3f\n", B[i]);
           }
           System.out.println();
    }
    /** function to print solution **/
    public void printSolution(double[] sol)
    {
        int N = sol.length;
        System.out.println("\nSolution : ");
        for (int i = 0; i < N; i++)
            System.out.printf("%.5f mm ", sol[i]*1000);
        System.out.println();
    }

    public RealVector getSolution() {
        return solution;
    }

}
```

D. Simulation.java

```
public abstract class Simulation {

        public String name; //Simulation name.
        public ExecutorService service;
        public PlotGraph plot;

        protected int n = 0;
        protected ArrayList<Future<EventSimulation>> eventSims;
        private int simulationLoops;
        protected int currentLoop = 0;
        protected boolean simStarted = false;
        private double histStart, histEnd;
        private int bins;

        public double smear = 0;

        public Simulation(String name, int simulationLoops, int bins, double histStart,
            double histEnd) throws Exception {
                this(name);
                this.name = name;
                this.simulationLoops = simulationLoops;
                this.bins = bins;
                this.histStart = histStart;
                this.histEnd = histEnd;
                if(simulationLoops <= 0) throw new Exception("Number of Simulation loops
                    can't be 0 or less!");
                eventSims = new ArrayList<Future<EventSimulation>>();

        }

        protected Simulation(String name) throws FileNotFoundException {
                System.out.println("Performing Simulation of "+name);

                //Import CSV file and select event to graph.
                EventsReader.init();

                //Set up Executor to multi-thread the simulation.
                service = Executors.newCachedThreadPool();
        }

        //Starts simulation and holds until Simulation complete.
        public void start() throws Exception {
                long time1 = System.currentTimeMillis();
                simStarted = true;
                while(currentLoop < simulationLoops) {
                        n = 0;
                        System.out.println("Loop "+currentLoop);

                        //Loop over every event
                        EventsReader.getEvents().values().forEach(event -> {
                                event.setup();
                                Future<EventSimulation> f = service.submit(() -> {
                                        n++;
                                        //Print out simulation progress.
                                        if(n % 250 == 0)
                                        System.out.println("Simulating event
                                            "+n+"/"+EventsReader.getEvents().size());
```

```java
                                return EventLoop(event.getId(),currentLoop);
                        });

                        eventSims.add(f);
                });

                //Hold until complete.
                while(!allTasksComplete(eventSims));
                System.out.println("Simulation Finished");
                postSimulation(currentLoop);
                eventSims = new ArrayList<Future<EventSimulation>>();
                currentLoop++;
        }
        postSimulationLoop();
        long finalTime = System.currentTimeMillis()-time1;
        System.out.println("Running time: "+finalTime/1000.0+"s");
        //events.events = null;
}

/**
 * Called after every Loop
 *
 * @param event
 * @return EventSimulation
 * @throws Exception
 */
public abstract EventSimulation EventLoop(int eventId, int currentLoop) throws
    Exception;


/**
 * Called after each simulation has finished.
 * @param currentLoop
 * @throws Exception
 */
public abstract void postSimulation(int currentLoop) throws Exception;

/**
 *  Called after all simulation loops have completed.
 */
public abstract void postSimulationLoop();


/**
 * @return GraphValues object that contains all
 * the information to be plotted onto the final graph.
 */
public abstract GraphValues configureGraph();

/**
 * @param point
 * @param sim
 * @return double - Return a value to add to histogram.

 */
public abstract double calculateHistogramValue(RealVector point, EventSimulation
    sim);

public void plotGraph() throws Exception {
        if(!simStarted) throw new Exception("Simulation must be run with start()
            before plotting graph");
        GraphValues vals = configureGraph();
        plot = new PlotGraph(vals.xVals,vals.yVals);
        plot.setGraphTitle(name);
        plot.setErrorBars(0, vals.Errors);
        plot.setXaxisLegend(vals.xAxis);
        plot.setYaxisLegend(vals.yAxis);
        plot.setLine(0);
        plot.plot();
}

public void saveRawDataToCSV(String filename) throws Exception {
        if(!simStarted) throw new Exception("Simulation must be run with start()
            before saving data");
        GraphValues vals = configureGraph();
        FileWriter file = new FileWriter(filename);     // this creates the file
            with the given name
PrintWriter outputFile = new PrintWriter(file); // this sends the output to file1

// Write the file as a comma seperated file (.csv) so it can be read it into EXCEL
outputFile.println(vals.xAxis+"("+vals.xUnit+")"+ "," +
    vals.yAxis+"("+vals.yUnit+")" + ", Error");

// now make a loop to write the contents of each bin to disk, one number at a time
// together with the x-coordinate of the centre of each bin.
for (int n = 0; n < vals.xVals.length; n++) {
    // comma separated values
    outputFile.println(vals.xVals[n] + "," + vals.yVals[n] + "," + vals.Errors[n]);
}
outputFile.close(); // close the output file
}

public void shutdown() {
        service.shutdown();
        try {
                service.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
        }
}

public <T> boolean allTasksComplete(List<Future<T>> f) {
        boolean allDone = true;
        Iterator<Future<T>> iter = f.iterator();
        while(iter.hasNext()) {
                Future<?> future = iter.next();
                if(!future.isDone()) {
                        allDone = false;
                        break;
                }
        }
        return allDone;
}

private Histogram getHist(double smear) throws Exception {
        Histogram hist = new Histogram(bins,histStart,histEnd, "Decay Length -
            "+smear);

        for(Future<EventSimulation> f : eventSims) {
                EventSimulation sim = f.get();
                RealVector p = findNearestPoint(sim,smear);
                double val = calculateHistogramValue(p, sim);
                if(p.getDimension() == 3) hist.fill(val);
        }
        return hist;
}

public double[] calculateRegression(double smear) throws Exception {
        Histogram hist = getHist(smear);
        return calculateRegression(hist);
}

public double[] calculateRegression(Histogram hist) {
        Regression reg = new
            Regression(hist.getX(),hist.getContent(),hist.getError());
        double[] result = new double[2];
        try {
                reg.exponentialSimple();
                result[0] = -reg.getBestEstimates()[0];
                result[1] = reg.getBestEstimatesErrors()[0];
                if(result[1] > 100 || reg.getDegFree() < 20) {
                        result[0] = 0;
                        result[1] = 0;
                }
        } catch(Exception e) {
                result[0] = 0;
                result[1] = 0;
        }
        return result;
}
```

```
        public RealVector findNearestPoint(EventSimulation event, double smear) throws
            Exception {

                //Fit straight lines to points - and check if sufficient data to fit one.
                ArrayList<StraightLineFactory> factories = new
                    ArrayList<StraightLineFactory>();
                for(ArrayList<RealVector> v : event.getSmearedDetections(smear)) {
                        if(!v.isEmpty()) {
                                StraightLineFactory line = new StraightLineFactory(v);
                                if(line.isValid()) {
                                        factories.add(line);
                                }
                        }
                }

                //Setup up line data to solve for nearest point.
                int n2 = factories.size();
                RealVector[] a = new RealVector[n2];
                RealVector[] d = new RealVector[n2];

                //Separate out each origin and direction vector.
                for(int i = 0; i < factories.size(); i++) {
                        StraightLineFactory line = factories.get(i);
                        a[i] = line.getOriginVector();
                        d[i] = line.getDirectionVector();
                }

                //Find the nearest point to all lines.
                FindNearestPoint p = new FindNearestPoint(a, d, 3);

                return p.getPoint();
        }

}
```

E. DetectorSmearingSimAveragePoint.java

```java
public class DetectorSmearingSimAveragePoint extends AveragePointSimulation {

        public DetectorSmearingSimAveragePoint(int eventId, int repeatMeasurements, int n,
            double start, double step)
                        throws Exception {
                super("Smearing", eventId, repeatMeasurements, n, start, step);
        }

        public static void main(String[] args) throws Exception {
                int eventID = 114;                                              //Selected
                    Event to simulate
                int numberOfRepeatSims = 1000;                      //Number of times to
                    simulate each event (Higher provides more accuracy)
                int numberOfSteps = 100;                                    //Number of steps to
                    take (Total simulations ran = numberOfRepeatsSims*numberOfSteps
                double startSmear = 0.0001;                         //Starting Detector Thickness
                double step = 0.05;                                         //Step to increase
                    the Detector thickness by each loop.

                DetectorSmearingSimAveragePoint sim = new
                    DetectorSmearingSimAveragePoint(eventID, numberOfRepeatSims,
                    numberOfSteps, startSmear, step);
                sim.start();
                sim.plotGraph();
                sim.saveRawDataToCSV("SmearingSimAveragePoint.csv");
                sim.shutdown();

        }

        @Override
        public void eventManipulation(EventSimulation sim, int n) {
                this.smear = start*Math.pow(10, -n*step);
                //System.out.println("Smear: "+smear+" n:"+n);
        }

        @Override
        public void postSimulation(int currentLoop) throws Exception {
                System.out.println("Smear: "+smear+" n:"+currentLoop);
                Value val = new Value(calculateRegression(hist));
                val.setX(smear);
                val.print();
                averageDistances.add(val);
                hist = getHist();
        }

}
```