PHYS488: Week3 - Objective-orientation introduction

Zachary Humphreys
200951438

February 15, 2017

Abstract

1 Task 1

This task was a step by step introduction the ideas of object oriented programming and modular code. It did this by requesting the addition of several methods of increasing complexity and then later having problems in the code be solved by calling them within or as an extension of the class.

The constructor is in *Histogram.java* to allows it to be called in *MakeHistogram.java* by line 23 where it creates a new instance of the object any parameters necessary, in this case, the size, and the minimum and maximum bin values as arguments.

```
Histogram hist = new Histogram(20, 0, 1);
```

which runs gives access to all the code Histogram and treats it as a black box, putting numbers in with it returning the required functionality without it being expressly coded into *MakeHistogram.java* file.

This approach allows creation of generic code which can be called and used when required.

The first addition to the code was a class member variable called **nfilled** and initialized to zero. This was to count how many times the method *fill()* was called. This required two additions, the first being at the top of the class, outside of any methods, to create the class member variable that any method could reference or access. The second line required it to be incremented upon each access to *fill()* and was done by adding the line

```
nfilled++;
```

to the last line outside of the *if else* function completely to ensure it is always incremented regardless of if the value is out of bounds of the histogram or not.

The class member variables **underflow, overflow,** and **nfilled** then needed to have access coded into the method. This was done building a getter method for each, with every one having a very similar structure to that of *getSize()*. They typically had this structure:

```
public int getNfilled()
 {
  return nfilled;
 }
```

with *nfilled* and the variable type being swapped out of whatever was required for that line of code. The code is self-explanatory simply returning the value of the variable, but is required as all the variables are given the tag, **private** which only allows the *Histogram.java* class to access them, but with these methods, *MakeHistogram.java* can now read these variables, but not write to them which is important for not changing the core functionality of *Histogram.java*.

The creation of a *getBinError* was an evolution of the previous task, but returning the solution to a small equation instead of just an already defined variable and was calculated using:

$$\Delta N = \sqrt{N} \tag{1}$$

where ΔN is the error on the number of counts within a bin of the histogram and N is the number of counts and was done with the method which took the input for the bin number that needed to be calculated for:

```
public double getBinError(int bin)
{
return Math.sqrt(hist[bin]);
}
```

In a similar sense to the last one, the next part was the same principle, but increasing the difficulty again. This required the implementation of a *getIntegral()* method. I chose to do this as a iterative function with the following code:

```
 public long getIntegral()
  {
  long sum=0;
  for(int i= 0; i< SIZE; i++)
   {
    sum = sum + hist[i];
   }
  return sum;
  }
```

where it would add all the values of each bin from the lowest to the highest.

The last subsection was a slightly different requiring the making of the method *print()* which called be called to print out the results of the class and would print out the contents of the bins, the number of times the histogram has been filled, the sum of all the bins, and the over/underflows. This method, similar to the one I implemented in my own code last week, takes a lot of the messy line printing code out of the *main* method of the *MakeHistogram.java* program and instead keep it as a method within the instance of the *Histogram* class itself. This was done with the following lines of code: which would loop for all the bin numbers and output it's number and contents, then on separate lines, would call the earlier methods to collect the needed values for the text. This method was called in *MakeHistogram.java* using

```
hist.print();
```

which would run all the code for that section and print to the console.

A typical output for this program would can be found in the appendix.

2 Task 2

This task required the modification and addition of the method *writeToDisk* (which was taken from a class used in last week's task) to be placed inside of the *Histogram.java* class as an additional bit of functionality. However, due to the current structure of the class, all of the variables required for it to function are already global variables that can already be accessed without the requirement of arguments, with the exception of the name of the file to be outputted, which remained as

A)

```
public void print()
{ //Prints relavent histogram information to console
 for(int i= 0; i< SIZE; i++)
 {
  System.out.println("Bin " +i+ " contents is: " +hist[i]);
 }
 System.out.println("Number of histogram fill attempts: "+getNfilled());
 System.out.println("The sum of all the bins is: "+getIntegral());
 System.out.println("The overflows are: "+getOverflow()+" and underflows: "+getUnderflow());
}
```

B)

```
outputFile.println("Trials , " + getNfilled());
outputFile.println("Integral , " + getIntegral());
outputFile.println("Overflows , " + getOverflow());
outputFile.println("Underflows , " + getUnderflow());
outputFile.println("bin# , bin_centre , counts , error");
```

Table 1: A) The Code for Task 1's print() function. B) The extra added lines of code to writeToFile for convenience.

the only argument.

The histogram range for the output was modified by changing the initialisation parameters in *MakeHistogram.java* on line 23 by changing:

```
Histogram hist = new Histogram(20, 0, 1);
```

to

```
Histogram hist = new Histogram(20, 0.4, 0.9);
```

giving the new correct ranges for the histogram. At this point, I also added a few more lines to the *writeToDisk* method to add some extra functionality to the spreadsheet output that would help with graphing and understanding the data from the outputted .csv file later. These are shown in part 2 of Table1 with the first four lines just calling the data from each of the methods to be printed at the top of the file, and the last line was to add some column titles to the data that was being outputted. The histogram for this task can be found in the Appendix.

3 Task 3

For this task, a histogram had to be plotted for the *nearGauss()* method that had already been written and provided in the *MakeHistogram* program.

The first thing that had to be done was to call a new instance of the Histogram which was done with:

```
Histogram gaussHist = new Histogram(50, -2, 2);
```

Just below the instance creation of the original histogram for the random data. A size of 50 was chosen to give plenty of data points, and as the sigma value for the data was 0.5, around a mean of 0, the low and high values of the bins were taken as ± 2 to encapsulate almost all of the data points, but to still leave a few to fall into the overflow and underflow. The values for the mean and sigma around for the Gaussian data creator was done by adding an additional loop just below the *for* loop used to fill the random data histogram. It was done using the code:

```
//For Gaussian Graph
for (int i=0;i<trials;i++)
{
 double value = nearGauss(0,0.5);
 gaussHist.fill(value);
}
```

which called the *nearGauss()* method and gave it the required arguments as specified by the task and would loop this method filling in the histogram for the total of number of trials specified by the input from the user.

The *nearGauss()* manages to give a (nearly) Gaussian curve by taking 12 random numbers, summing them all together and then using the mean and sigma value arguments to scale the sum's value correctly upon the curve. This summation tends to Gaussian around a single value due to the effect of averaging the value of random numbers will converge on the on the midpoint between the limits such that:

$$\lim_{n\to\infty} \frac{1}{n} \cdot \sum_{i=1}^{n} x_i = \frac{b-a}{2} \quad (2)$$

where x is a random number between limits b and a.

The data was exported to both the console and a file through the use of the lines of code:

```
gaussHist.print();
gaussHist.writeToDisk("gauss_test.csv");
```

Which call the methods print() and writeToDisk() of the instance and print to file and console respectively.

The graph can be found in the appendix.

4 Task 4

For the last task, both the method for generating the data, and the method for filling the histogram were needed to be written. The data points needed to be calculated with:

$$D = -C \cdot ln(r) \quad (3)$$

where D is the datapoint, C is a constant of value 15, and r is a random value between 0 and 1. To calculated these datapoints, the following code was written:

```
for(int i=0;i<trials;i++)
{
 double value = (-15*Math.log(randGen.nextDouble()));
 dHist.fill(value);
}
```

which would generate values for each trial number, and fill the histogram instantiated by:

```
Histogram dHist = new Histogram(50, 0, 150);
```

where dHist is the name of this histogram. The values this histogram would generate are generally between 0 and a reasonable approximate maximum of 150 for $r = 0.0001$. 50 bins were chosen to give many data-points to demonstrate a smooth curve. The data was outputted to a file in the same manner as earlier using:

```
dHist.writeToDisk("d_test.csv");
```

and the graph can be found in the appendix.

5 Appendices

5.1 Task 1 Console Output

```
Bin 0 contents is: 25022
Bin 1 contents is: 24684
Bin 2 contents is: 25001
Bin 3 contents is: 25150
Bin 4 contents is: 24771
Bin 5 contents is: 24981
Bin 6 contents is: 25139
Bin 7 contents is: 25071
Bin 8 contents is: 24891
Bin 9 contents is: 25062
Bin 10 contents is: 25245
Bin 11 contents is: 24898
Bin 12 contents is: 24879
Bin 13 contents is: 25270
Bin 14 contents is: 25165
Bin 15 contents is: 24862
Bin 16 contents is: 24879
Bin 17 contents is: 24943
Bin 18 contents is: 25236
Bin 19 contents is: 25195
Number of histogram fill attempts: 1000000
The sum of all the bins is: 500344
The overflows are: 99719 and underflows: 399937
```

5.2 Task 2: Random-data Histogram

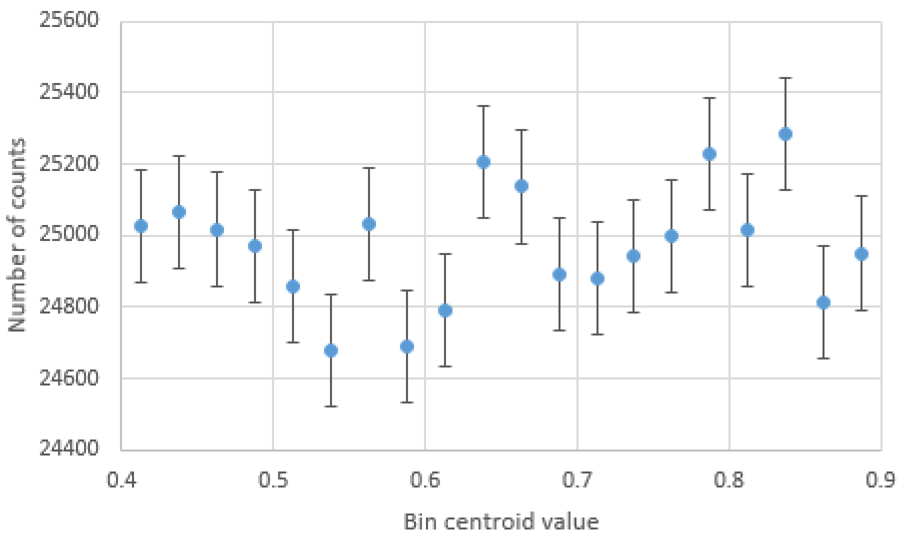

Figure 1: This is the graph generated by the code set for task 2 for a set of 1 million trials.

5.3 Task 3

5.4 Task 4

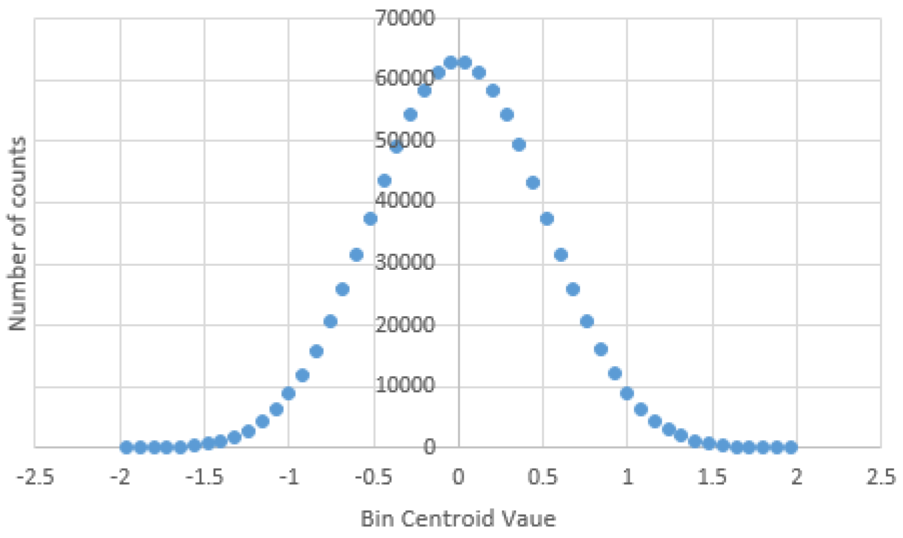

Figure 2: This is a graph of the data from the Gaussian data generator. The errorbars are very small and cannot be seen as it was run for 1 million trials

```
Binlow  −2
Binint  0.08
nbins   50
Trials  1000000
Integral  999978
Overflows  14
Underflows  8
bin#   bin_centre   counts   error
0  −1.96  17  4.123105626
1  −1.88  38  6.164414003
2  −1.8  76  8.717797887
3  −1.72  114  10.67707825
4  −1.64  225  15
5  −1.56  417  20.42057786
6  −1.48  703  26.51414717
7  −1.4  1160  34.05877273
8  −1.32  1878  43.33589736
9  −1.24  2860  53.47896783
10  −1.16  4395  66.29479618
11  −1.08  6467  80.41765975
12  −1  8806  93.84028985
13  −0.92  11997  109.5308176
14  −0.84  15917  126.1625935
15  −0.76  20584  143.4712515
16  −0.68  25945  161.0745169
17  −0.6  31457  177.3612133
18  −0.52  37375  193.3261493
19  −0.44  43666  208.9641118
20  −0.36  48975  221.3029598
21  −0.28  54498  233.448067
22  −0.2  58299  241.4518586
23  −0.12  61308  247.6045234
24  −0.04  62731  250.4615739
25  0.04  62909  250.8166661
26  0.12  61163  247.3115444
27  0.2  58227  241.3027144
28  0.28  54225  232.8626204
29  0.36  49506  222.4994382
30  0.44  43240  207.9422997
31  0.52  37348  193.2563065
32  0.6  31343  177.0395436
33  0.68  26040  161.369142
34  0.76  20640  143.66628
35  0.84  16169  127.1573828
36  0.92  12168  110.3086579
37  1  8893  94.3027041
38  1.08  6340  79.62411695
39  1.16  4309  65.64297373
40  1.24  2945  54.2678542
41  1.32  1931  43.94314509
42  1.4  1139  33.74907406
43  1.48  680  26.07680962
44  1.56  396  19.89974874
45  1.64  236  15.3622915
46  1.72  130  11.40175425
47  1.8  52  7.211102551
48  1.88  29  5.385164807
49  1.96  12  3.464101615
```

Figure 3: This is the data from the csv file generated.

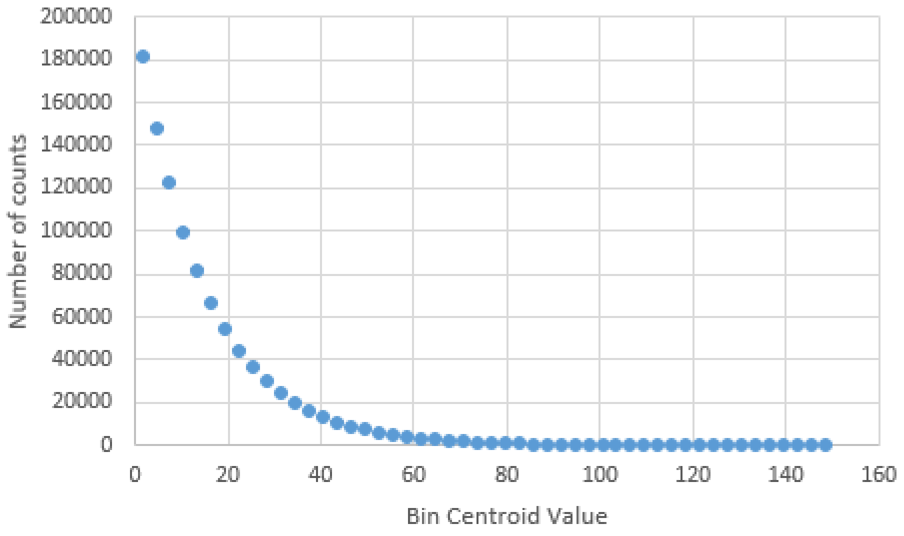

Figure 4: This is the graph generated by the equation given in task 4 for 1 million trials. Again, the errors are too small to be seen on the graph.