PHYS488: Week1 - General Excerises

Zachary Humphreys
200951438

30/1/2017

Abstract

This weeks set of tasks focused on learning about loops, if functions, and arrays, and involved executing some sample programs provided for Task 1, updating a program created last week to be done using arrays, and loops in Task 2, and learning how to export data from code by writing to an .csv file in Task 3.

1 Task 1

In this task, five java programs were provided and each of these codes were to be run and inspected to see how they work. These programs were imported into BlueJ and, in the case of *FindRoots.java*, corrected and executed.

1.1 SimpleSum.java

This program was created to demonstrate the overall structure of how Java programs should be written, and specifically to demonstrate how to implement methods within a class.

Lines 9 through 16 show a simple method that takes two *double* variable inputs and outputs another *double* variable with the value of the two inputs added together as it returns *sum* which is a variable stated and calculated with lines 12 and 13.

The main method then proceeds with asking the user to input two different variables into the console then uses the method from earlier in line 25 to calculate the sum of these two variables with:

```
double ans = getsum(first , second);
```

which returns the sum and in the subsequent line, this is printed to the user in the console

1.2 FindRoots.java

This program iterated on the demonstrations shown in *SimpleSum.java* by having a similar structure of one short method and a main method handling most of the work.

The method, like previous, takes two *double* argument (inputs) and returns (outputs) a *double*. However, the method is slightly more complicated than the previous program: instead of simply summing the two inputs, it finds the n^{th} root of the x input by returning the exponential of line 15.

There are four provided examples of this method being called in the main method, with the last one failing to work (line 27):

```
screen.println(" The 3/2  root of 9*9*9 =
" + nthRoot(9*9*9,3/2));
```

which returns the value 729, as if no n^{th} root had been applied. This is due to the fact that the n given in the function is 3/2 where java treats both values as integers, hence returning the value 1. By simply changing it to (3./2), Java no longer treats the 3 as an integer and it returns the correct value, same as the line of code above it.

1.3 SimpleLoops.java

This code is different from the other two as it only consists of a single main, though it has two separate functions within it. The first function demonstrates a *for* loop for a loop where the number of iterations to be done is well known. It's contained within lines 14 to 16 and simply takes the iterations number, *n*, and raises it to the power of itself.

The second section in the main simply demonstrates how to use a *while* loop to produce a simple number guessing game that keeps looping until the user's answer is equal to the randomly generated number in line 23. Within this while loop it has an *if/else* statement between lines 28 and 32 that finishes the print line from above with either higher or lower depending on whether the guess is higher or lower respectively.

If the user finally inputs the correct value, the while loop will stop and print a line congratulating the player. Unfortunately, if the player is not good at guessing, it could potentially form an infinite loop.

1.4 SimpleArray.java

This program demonstrates a way to input an array of a length $numberToStore$ determined by the user. It then uses this value to determined the maximum number of loops in the *for* loop starting on line 16, which asks the user to input each number followed by an enter in turn to be put into the array. The second *for* loop is used to output each of the numbers from the compiled array to check if these were the values the user wanted.

1.5 Interpolate.java

The overall function of this program is to take an input from the user and output an estimated reaction cross section sigma value. It did this by first having an array of values for the cross section in line 14. These effectively provided a framework for the curve by having the values calculated at exact energy intervals of 10 MeV. The while function on lines 28 through 54 would loop as long as the user did not input a zero, and would ask the user to input an energy.

The energy inputted by the user would then have it's array element number calculated on line 36 with lines 39 to 42 doing a quick check to make sure the value inputted actually fell into the energy range of a bin.

Lines 44 to 48 gave the bin's element number, and the energy range it fell into, and lines 51 to 53 then did a quick interpolation by first finding how close it was the top end of the bin, and using this weighting in line 52 to do a weighted average of the cross section values assosiated to the energy values of the bin and then outputted this weighted average sigma value to the console in line 53.

2 Task 2

2.1 Methodology

This task involved modifying the code for the last section of week 1's set of tasks to reduce the number variables and replace them with arrays instead, and to reduce repetition in the code and replace repeatable bits with loops instead.

To first start this task, I started by writing down on paper in "pseudo-docode" what I wanted my program to do, and how to do it. The initial ideas, though almost identical in structure to the final implementation, was very different in the techniques used as new and alternative techniques were learnt in the few days trying to achieve the program I envisioned.

This meant a fair amount of extra knowledge not contained within the lecture had be learnt and adapted to get my code to work how I wanted it to, and this was done mostly through the use of recommended textbook [1] and Oracle's own documentation online concerning their code. The initial paper plan for the program went thus:

```
Class 4Vecs V2
  Method: 4Vec ToArray        (1)
  Method: 4Vec MomentumCalc   (2)
  Method: 4Vec MassCalc       (3)
  Method: 4Vec Adder          (4)
  Main:     Loop(n from 0 to 1 ) {
            Input 4Vec_n as seperate variables
      Do methods (1), (2), (3) }
      Use (4) to add 4Vec_0 and 4Vec_1
      Do (2) (3)
      Output answers to console - END
```

Unfortunately, after a few hours with this plan, two big problems occurred: Issues using two arrays as arguments for methods, and getting a loop to produce variables with a suffix "$_n$" where n was the loop number which could be used for calculations in the loop. Another problem was the clumsy unclear main, filled with more complex code than some of the methods making it hard to understand and read. Instead, this iteration was almost entirely scrapped with only some of the shells of the methods intact, such as the discovery that it was much easier to implement a method to calculate both the Momentum and the invariant mass of the given four vector, and was much fewer lines of code as a result.

As a by-product of learning about multiple arrays as arguments to a method, 2D arrays were discovered, and from there, a new cleaner strategy was implemented. The paper code was rewritten and revisualised for clarity and the final paper iteration for the framework was thus:

```
Class 4Vecs V2.1
  Method: 4Vec Array Builder (1)
  Method: 4Vec Mass and Momentum Extractor (2)
  Method: 4Vec Array Adder (3)
  Main:    4VecArray = (1)
      MassMomentumArray = (2)
      4VecAddedArray= (3)
      AddedMassMomentumArray = (2)
      Print needed stuff - END
```

This iteration, at first, still contained a mix of 1D and 2D arrays for calculations, but eventually, the entire framework was readjusted to accepted 2D arrays at every input and output for all the methods for consistency. [2]

2.2 Method 1 - Array-Compiler

The first method, the Array Builder (lines 15 to 32), was created to take a string of numbers from the user and convert them into usable *double*s for calculations later on, built into a 2D array, where each array row was a four vector. As the question only required

[1] J. Hubbard, Programming with Java,(2004), 2nd edition, McGraw-Hill.

[2] The history of this program during the later stages can be viewed at `https://github.com/donmegamuffin/Phys488_Workshop2`

calculation for two four vectors, the for loops number of loops are baked in to have them. The *for* loop on line 20 deals with the four-vectors as a whole, where it requests the user to input into the console the four vector in the format:

```
        000 000 000 000
e.g.    150 040 -35 -20
```

and the second *for* loop would pull substrings out of the input string and convert them into *double* array elements.

This had a couple of limitations: all values had to be integers, positive values could only go up to a maximum of 999 and negatives were limited to -99 as - took up a character slot. This was due to the fact I could not find out how to code it to read a number up to an empty space and then turn everything up to then into a number, to then continue onwards to the next number, etc. However, for the values requested in the Task, this was more than sufficient.

The output of this method was an output of a 2D Array with each vector stored in one row.

2.3 Method 2 - ArrayMandPExtractor

This method, took one of the four-vector 2D arrays, and would then calculate the absolute momentum and invariant mass (lines 40 and 41) for both vectors, and place them into a 2x2 2D matrix from earlier, again, with each vectors values placed in separate rows.

2.4 Method 3 - ArrayAdder

Again, this is another simple Method that initialises a two row, four column array and takes the input four vector array, and adds the values from row 2 to row 1 and outputs this new 2D array.

2.5 Method 4 - ConsoleWriter

To make the main method more readable, all the code that was used to write the solutions to the console was moved to a separate method which takes 3 arguments that are all of the 2D matrices generated so far. This was done with a series of *printf* commands to enable the rounding of the *double* values that will be inputted into it using the string "$\%.3f$" inbedded into the line of code which gave the value to 3 decimal places, instead of the long string of numbers it would generate by using a *println* command instead. This required quite a few lines as a method to have multiple variables put into a single *printf* command with the rounding included could not be found.

For debugging purposes, a simple method for outputting each of the matrices directly to the console had be found, and it was found that it was possible to use:

```
import java.util.Arrays;
...
...
screen.println(Array.deepToString())
```

to print the arrays to the console in as few lines as possible, but unfortunately doesn't give any context to the numbers, so was replaced after the debugging.

2.6 Main

This section is now much cleaner than the original implementation consisting of only 5 lines of code, 4 of which (lines 82 to 85) are simply 2D array definitions with the final one of these (line

Task3.1

```
41 final double binlow  = 0.4; // this is the low edge of the first bin , hist1[0].
42 final double binhigh = 0.9; // this is the upper edge of the last bin, hist1[SIZE-1].
43 long numberUnderflows=0; // this is the number of random values that fall below binlow.
44 long numberOverflows=0; // this is the number of random values that fall above binhigh.
48 double [] histError = new double[SIZE]; // Empty array is filled zeroes
69  if(randNumber <=binlow || randNumber >=binhigh) //
70  {
72   if(randNumber <=binlow)
73   {
74     numberUnderflows++;
75    }
76   if(randNumber >=binhigh)
77   {
78     numberOverflows++;
79   }
80  }
91 histError[bin] = Math.sqrt(hist[bin]);
92 screen.printf("Bin number " + bin + " contents =  " +
hist[bin]+ ";error = " + "%.3f", histError[bin]);
93 screen.println(" ("+ (100*histError[bin]/hist[bin]) +"%)");
97 screen.println("The number of random numbers overflowed: " +numberOverflows);
98 screen.println("The number of random numbers underflowed: " +numberUnderflows);
```

Task3.2

```
21  outputFile.println("ntrials   , " + trials);
22  outputFile.println("#Underflowed   , " + under);
23  outputFile.println("#Overflowed   , " + over);
31  outputFile.println(n + "," + binCentre + "," + hist[n] +"," + Math.sqrt(hist[n]));
100 writeToDisk(hist, binlow, binsize, trials, numberUnderflows, numberOverflows, "test.csv");
```

Table 1: These are the added lines of code for Tasks 3.1 and 3.2

86) taking these arrays and applying them to the ConsoleWriter method. This method is, as a result, now much easier to read and understand as much of it is written in almost plain English.

3 Task 3

This section of the tasks involved editing the functionality of the program *GenerateHistogram.java* to fit the new requirements specified in each section. After all of the adjustments were made, the exported data was used in excel to produce a Histogram graph of the random numbers.

3.1 Task3.1

This task required the changing of the histogram range from $0 < r < 1$ to $0.4 < r < 0.9$ by changing the binlow and binhigh values (lines 42 and 43). This initially seemed like a simple task but unfortunately caused errors when it would attempt to assign a value to a bin that did not exist (as it was out of the range) causing it to over or underflow. This was corrected by having an if statement that checked each value before placing it into a bin (lines 69 to 80) and if the random number was outside of the range, it would place it into the over or underflow section respectively.

The next task was to calculate the error of each bin. To do this I created an array of the exact same size as the $int[]hist$ on line 48 and then calculated the error for each of these bins using the formula:

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

where N is the number of counts within a bin, and ΔN is the error on these counts. This was done on line 91 and printed as a value on line 92 and as a percentage on line 93, done by making a few additions such as changing *println* to *printf* and adding the needed variables and text. The percentage error was calculated trivially as:

$$\Delta N(\%) = \frac{100 \cdot \Delta N}{N} \qquad (2)$$

The console output for this task is provided in the Appendix.

3.2 Task3.2

In this task, the method *writeToDisk* has to be slightly adjusted as to output a few extra lines at the top of the .csv file to print the number of trials, overflows, and underflows. This was done in lines 21 to 23 with a few simple additions following the the same syntax as the lines above. The other task was to add an extra column to csv to give the statistical error on the bin which was done in line 31 where the $+","+Math.sqrt(hist[n])$ addition adds this feature and would be done for each bin as it was in the *for* loop of line 26.

For the method to read all these values for it to output it to a file, extra arguments were created and inputted from the main method, and this can be seen in line 10 and line 100.

3.3 Task3.3

This section was to simply open the .csv file produced by the code, and use it to produce a "graph of the histogram" with the error bars for the y values generated by the code. I produced two graphs, one of a typical barchart and another of a point graph, as I was unsure if the task wanted a "histogram graph" or simply a graph that represented the data. These can be found in the appendix along with a sample csv output.

4 Appendices

4.1 Task2: Console Output

CASE 1:

```
FOR FOUR VECTOR: 1
Please input the four vector where each four vector has 3 digits and each is separated by a space:
005 004 000 000
FOR FOUR VECTOR: 2
Please input the four vector where each four vector has 3 digits and each is separated by a space:
005 -04 000 000
The momentum of Four Vector 1 is 3.000 GeV and the rest mass is 4.000 GeV.
The momentum of Four Vector 2 is 3.000 GeV and the rest mass is 4.000 GeV.
The addition of the two vectors is (10.0,0.0,0.0,0.0).
The momentum of Added Four Vector is 10.000 GeV and the rest mass is 0.000 GeV.
```

CASE 2:

```
FOR FOUR VECTOR: 1
Please input the four vector where each four vector has 3 digits and each is separated by a space:
150 040 -35 -20
FOR FOUR VECTOR: 2
Please input the four vector where each four vector has 3 digits and each is separated by a space:
250 -40 060 070
The momentum of Four Vector 1 is 138.834 GeV and the rest mass is 56.789 GeV.
The momentum of Four Vector 2 is 228.910 GeV and the rest mass is 100.499 GeV.
The addition of the two vectors is (400.0,0.0,25.0,50.0).
The momentum of Added Four Vector is 396.074 GeV and the rest mass is 55.902 GeV.
```

4.2 Task2: Code

```java
/*Author: Zachary Humphreys; ID: 200951438
*Date: 07/02/17
*/
import java.io.*;
import java.util.Arrays;

class FourVecV2
{
 static BufferedReader keyboard = new
 BufferedReader (new InputStreamReader(System.in)) ;
 static PrintWriter screen = new PrintWriter( System.out, true);

 // ----------- METHODS ----------------------------------------------------------------------------------------
 // Method 1: Compiles inputs from user into an array
 private static double[][] ArrayCompiler() throws IOException
 {
  //Asks user to input four vector into the form: 123 456 789 012
  //Pulls values for energy and momentum vectors out of the string and compiles them in an array
  double[][] arrayOut = new double [2][4]; //{{0,0,0,0},{0,0,0,0}};
  for(int i=0;i<2;i++)
  {   //Request input from user
   screen.println("FOR FOUR VECTOR: "+(i+1));
   screen.println("Please input the four vector where each four vector has 3 digits and each is separated by a space:\
   String userInput = new String(keyboard.readLine());
   for(int n=1; n <= 4; n++)
   {   //Pulls the numbers out of the string provided in triplets of characters
    int m = (n*4 - 3);
    arrayOut[i][n-1] = new Double(userInput.substring(m-1,m+2)).doubleValue();
   }
  }
  return arrayOut;
 }

 //Method 2: Takes an inputted Array and calculates the Rest Mass and Invariant Momentum
 private static double[][] ArrayMandPExtractor(double[][] arrayIn)
 {
  double[][] arrayOut = new double [2][2];
  for(int i=0;i<2;i++)
  {   //Does the maths to calculate |P| and M.
   double AbsMomentum = Math.sqrt(Math.pow(arrayIn[i][1],2) + Math.pow(arrayIn[i][2],2) + Math.pow(arrayIn[i][3],2));
   double InvariantMass = Math.sqrt(Math.pow(arrayIn[i][0],2) - Math.pow(AbsMomentum,2));
   //Files the |P| and M values into the i'th row of the new 2x2 array
   arrayOut[i][0] = InvariantMass;
   arrayOut[i][1] = AbsMomentum;
  }
  return arrayOut;
 }

 //Method 3: Takes a 2D array and adds  the two dimensions and puts them in top row
 private static double[][] ArrayAdder(double arrayIn[][])
 {
  double[][] arrayOut = new double [2][4];
  for(int n = 0; n<4; n++)
  {
   arrayOut[0][n] = (arrayIn[0][n] + arrayIn[1][n]);
  }
  return arrayOut;
 }

 //Method 4: Takes the input Arrays and outputs the data to the console as strings
 private static void ConsoleWriter(double[][] PM_Array, double[][] fourVadded_Array, double[][] PMadded_Array)
 {
  //For input arrays
  screen.printf("The momentum of Four Vector 1 is %.3f" , PM_Array[0][0]);
  screen.printf(" GeV and the rest mass is %.3f", PM_Array[0][1]);
  screen.println(" GeV.");
  screen.printf("The momentum of Four Vector 2 is %.3f", PM_Array[1][0]);
  screen.printf(" GeV and the rest mass is %.3f", PM_Array[1][1]);
  screen.println(" GeV.");
  //For added arrays
  screen.println("The addition of the two vectors is (" +fourVadded_Array[0][0]+ ","
 +fourVadded_Array[0][1]+","+fourVadded_Array[0][2]+","+fourVadded_Array[0][3]+").");
  screen.printf("The momentum of Added Four Vector is %.3f", PMadded_Array[0][0]);
  screen.printf(" GeV and the rest mass is %.3f", PMadded_Array[0][1]);
  screen.println(" GeV.");
 }

 // --------- MAIN ---------------------------------------------------------------------------------------
 public static void main(String[] args) throws IOException
 {
  //Calculates necessary Variables using methods
  double[][] fourVector_Array = ArrayCompiler();
  double[][] MassMomentum_Array = ArrayMandPExtractor(fourVector_Array);
  double[][] fourVectorAdded_Array = ArrayAdder(fourVector_Array);
  double[][] MassMomentumAdded_Array = ArrayMandPExtractor(fourVectorAdded_Array);
  //Uses Arrays to write to console
  ConsoleWriter(MassMomentum_Array,fourVectorAdded_Array,MassMomentumAdded_Array);

  //Crude output variables for debug purposes
  /*
  screen.println(Arrays.deepToString(fourVector_Array));
  screen.println(Arrays.deepToString(MassMomentum_Array));
  screen.println(Arrays.deepToString(fourVectorAdded_Array));
  screen.println(Arrays.deepToString(MassMomentumAdded_Array));
  */
 }
}
```

Please see `https://github.com/donmegamuffin/Phys488_Workshop2/blob/master/FourVecV2.java` if you would like a more clear copy of the code.

4.3 Task3.1: Console output Example

```
Input the number of random numbers to generate
500000
The 0th number is =  0.2598101114675839
The 25000th number is =  0.34673665069012183
The 50000th number is =  0.5851592697737925
The 75000th number is =  0.03252903391072359
The 100000th number is =  0.8008461936246785
The 125000th number is =  0.8676123416803804
The 150000th number is =  0.892947145222212
The 175000th number is =  0.7342759532590893
The 200000th number is =  0.5999943151166929
The 225000th number is =  0.033515866436045316
The 250000th number is =  0.16910984930762019
The 275000th number is =  0.9509342537333896
The 300000th number is =  0.2566711971832861
The 325000th number is =  0.11717044748210059
The 350000th number is =  0.4735076089357395
The 375000th number is =  0.16286303339506158
The 400000th number is =  0.07462787930866577
The 425000th number is =  0.6681433283087862
The 450000th number is =  0.2934794494368943
The 475000th number is =  0.006659936619997975
Bin number 0 contents =  12572;error = 112.125 (0.8918623156480929%)
Bin number 1 contents =  12619;error = 112.334 (0.890199876762518%)
Bin number 2 contents =  12642;error = 112.437 (0.8893897233781289%)
Bin number 3 contents =  12363;error = 111.189 (0.899369313562268%)
Bin number 4 contents =  12452;error = 111.589 (0.8961494529112316%)
Bin number 5 contents =  12434;error = 111.508 (0.8967978708178482%)
Bin number 6 contents =  12574;error = 112.134 (0.8917913837423341%)
Bin number 7 contents =  12708;error = 112.730 (0.8870771590205824%)
Bin number 8 contents =  12537;error = 111.969 (0.8931063702570722%)
Bin number 9 contents =  12724;error = 112.801 (0.8865192488024881%)
Bin number 10 contents =  12434;error = 111.508 (0.8967978708178482%)
Bin number 11 contents =  12302;error = 110.914 (0.9015963371316719%)
```

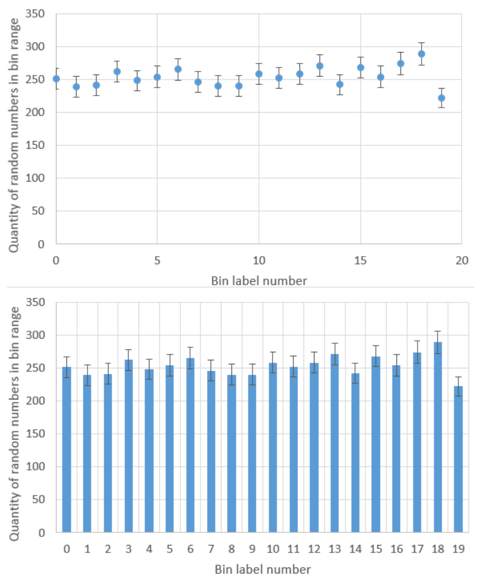

Figure 1: These are the graphs of the data generated from the csv file, both displaying the same data but in different graph styles.

```
Bin number 12 contents =  12426;error = 111.472 (0.8970865086981312%)
Bin number 13 contents =  12473;error = 111.683 (0.8953947400781449%)
Bin number 14 contents =  12495;error = 111.781 (0.894606130121642%)
Bin number 15 contents =  12387;error = 111.297 (0.8984976202985103%)
Bin number 16 contents =  12530;error = 111.937 (0.8933558064776197%)
Bin number 17 contents =  12367;error = 111.207 (0.899223855157053%)
Bin number 18 contents =  12361;error = 111.180 (0.8994420692403841%)
Bin number 19 contents =  12405;error = 111.378 (0.8978455111377533%)
Number of trials = 500000 , the sum of the contents = 249805
The number of random numbers overflowed: 49849
The number of random numbers underflowed: 200346
Writing to disk, please wait....
Data written to disk in file test.csv
```

4.4 Task 3.2 and 3.3: Graphs and .csv file output

```
Binlow   0.4
Binint   0.025
nbins    20
ntrials           10000
#Underflowed      3936
#Overflowed       990
0        0.4125   251      15.84297952
1        0.4375   239      15.45962483
2        0.4625   241      15.5241747
3        0.4875   262      16.18641406
4        0.5125   248      15.74801575
5        0.5375   254      15.93737745
6        0.5625   265      16.2788206
7        0.5875   246      15.68438714
8        0.6125   240      15.49193338
9        0.6375   240      15.49193338
10       0.6625   258      16.0623784
11       0.6875   252      15.87450787
12       0.7125   258      16.0623784
13       0.7375   271      16.46207763
14       0.7625   242      15.55634919
15       0.7875   268      16.37070554
16       0.8125   254      15.93737745
17       0.8375   274      16.55294536
18       0.8625   289      17
19       0.8875   222      14.89966443
```

This is the raw data outputted from the .csv file to generate the graphs above.