**PHY1003 – Assignment 3 – Random Number Generation and Stochastic Processes – Report**

The aim of this assignment is to explore a stochastic system with the aid of a random number generator. The function used as a random number generator is as shown:

where:

*seedn* is an integer number, and;

*seedn+1* is the new random number generated from *seedn*.

This type of generator is called a Linear Congruential Generator and each element affects the generation of a random umber in a specific fashion.

An initial program will be written around the above formula to create a random number generator which will be investigated within this report by using different values for *a, b* and *c.* A second program will be written to use this random number as a probabilistic determinant to create a movement, simulating a one-dimensional walk along an integer line. This program will then be adapted for use in three dimensions.

A three dimensional random walk can be used as a basic model for many natural phenomena, for example, the random motion of particles in fluids and gases.

Tasks 1 and 2: Investigating a simple random number generator

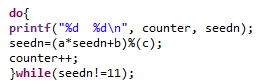
 A program was written using the function stated above to generate a random (or pseudo-random) number. The numbers generated which each iteration can give a useful insight into the workings of the function and the altering of *a, b* and *c* values can shed light on each values role in determining a random number.

Figure 1: do…while loop to iterate over formula

This program was coded using a ‘do…while’ loop rather than a standard ‘while’ loop as the first iteration initially fulfils the while condition which would end the loop prematurely. Thus, a do…while loop allows for at least one iteration to be carried out before the condition is checked against. A counter was also used and was output alongside the random number, which allows the user to see how many iterations have been carried out before the random number generator start repeating itself (the act of a random number generator repeating a pattern is somewhat against the idea of true randomness, which is why it was called a pseudo-random number generator previously. However, the randomness of this function is sufficient for the application).

As mentioned above, by varying *a*, *b* and *c* we can investigate how the randomness of the function is characterised. For the first and second versions of the program the variables were defined as:

*a1* = 147; *a2* = 8005;

*b1* = 55; *b2* = 1;

*c1* = 256 ( = 28); *c2* = 65536 (= 216)

The table below shows the output for the first five and final three output values.

*Table 1: Random numbers generated by two different sets of values for a, b and c*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration (*#*) | v.1 - Value Generated (*nx*) |  | Iteration (*#*) | v.2-Value Generated (*nx*) |
| 0 | 11 |  | 0 | 5427 |
| 1 | 136 |  | 1 | 58304 |
| 2 | 79 |  | 2 | 41665 |
| 3 | 148 |  | 3 | 15622 |
| 4 | 51 |  | 4 | 11423 |
| 5 | 128 |  | 5 | 18396 |
| 126 | 103 |  | 65534 | 34933 |
| 127 | 92 |  | 65535 | 62090 |
| 128 | 11 |  | 65536 | 5427 |

Please note that although the initial seed value for each set of results is different this has no effect on the generation of the points aside from the starting point of the formula, the number of iterations will not change. It has been said that the period is perhaps the most important factor for a random number generator (L’Ecuyer, 2017), we can see from the table above that the second set of variables has a much larger period, it can thus be said that the second set of variables produces a more robust random number generator.

Task 3: One-dimensional random walk

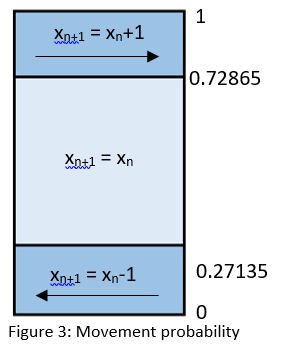
 This program is made to simulate a one-dimensional random walk. The program from Task 2 (the more robust RNG) is used to generate a random number which is then converted into a fractional number (called a *p*-value from here) between 0 and 1 by dividing the random seed by ‘*c*-1’ (Subtracting one is a more mathematical correct way of finding this probability as zero has been included as the first iteration, taking away 1 ensures that the probabilities output never exceed 1). Dependant on where this *p-*value lies between 0 and 1 will determine the movement made (Figure 3 outlines the probabilities used in this program and their associated movements). This technique of simulating the behaviour of a system which incorporates stochastic behaviour by using a randomness generator to vary the behaviour of the system is known as the Monte Carlo routine (Kaye, 1994). This process is also seen as a Markov chain.

Figure 2: Incorporating RNG into the program and finding its *p*-value

It should be noted that there are other ways to arrange the probabilities in figure 3. The probability of p+1 could be moved to 0.27135 < x < 0.5427 (or anywhere else on the diagram), this would affect the outcome of the movements for each random walk but would not alter the statistical outcome of the walk. In this investigation the quantitative side of the results is of less importance than the qualitative understanding of the data.

 The program iterates in a main loop over 1000 times, each of these is a complete walk and within this walk a second for loop is used to generate 100 steps. Note how at the beginning of each walk the value of x has been reset, this is to ensure that each walk starts from zero (ie. each walk is independent of the last walk). Once each step has been taken the value of x at that step is added into an array, as is the value of x2, where each array value is the sum of the positions at that same value of step (this means that all the step one’s from each walk are added together, all the step two’s etc.). Once all of the walks have been completed the average of each array slot is taken by dividing by the number of walks by the summed value (see Figure 5). The data generated can be used to plot two graphs, one of average position of each step (equating to the average random walk) and one of the x2 value against each step (Figures 6 and 7, respectively).

Figure 5: Averaging each array slot

Figure 4: For loops to iterate 100 steps within 1000 walks

Figure 7 shows the average walk of the 1000 walks generated. As there is an equal chance of the program adding 1 to x or minusing 1 it is expected that the average walk does not deviate far at any one point from 0. This can be seen in the figure below as the largest negative value is -0.264. Although the average walk is overwhelmingly negative and does not recross the x-axis after the third step it can be said that it is guarenteed that the walk will recross the x-axis. This is the same as saying that at some point the <x> value will be the same as a previous <x> value, this returning to a previous state means the Markov chain studied here is recurrent (this is the basis of the Gambler’s falacy).

Figure 7: The average position of each step of 1000 walks – an average walk

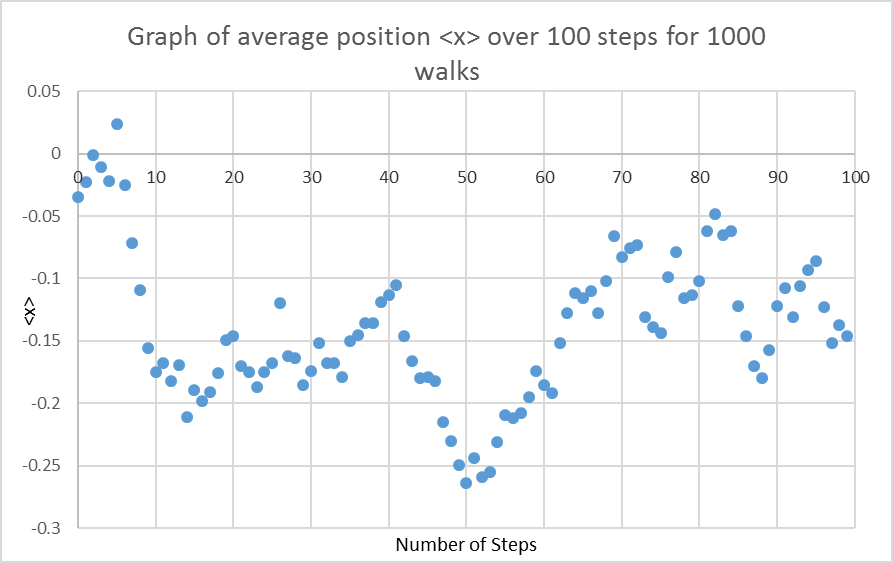


Figure 8: The average position squared of each step of 1000 walks

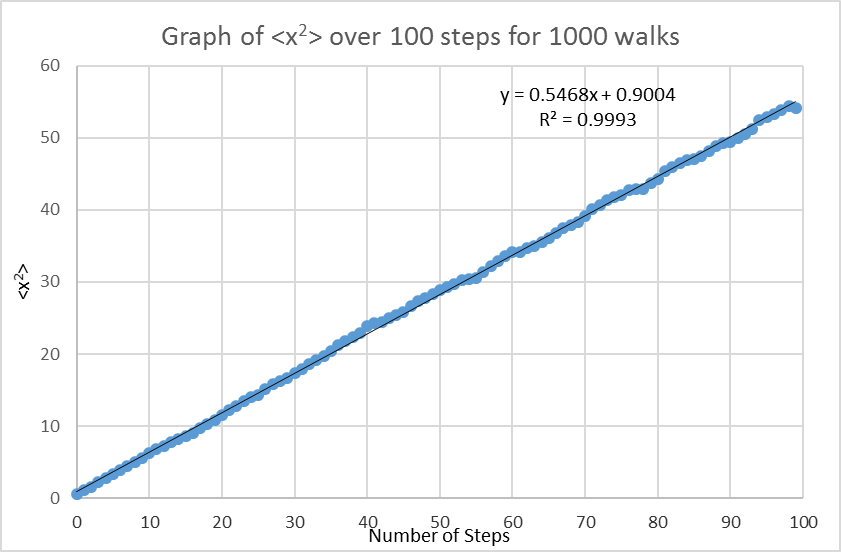


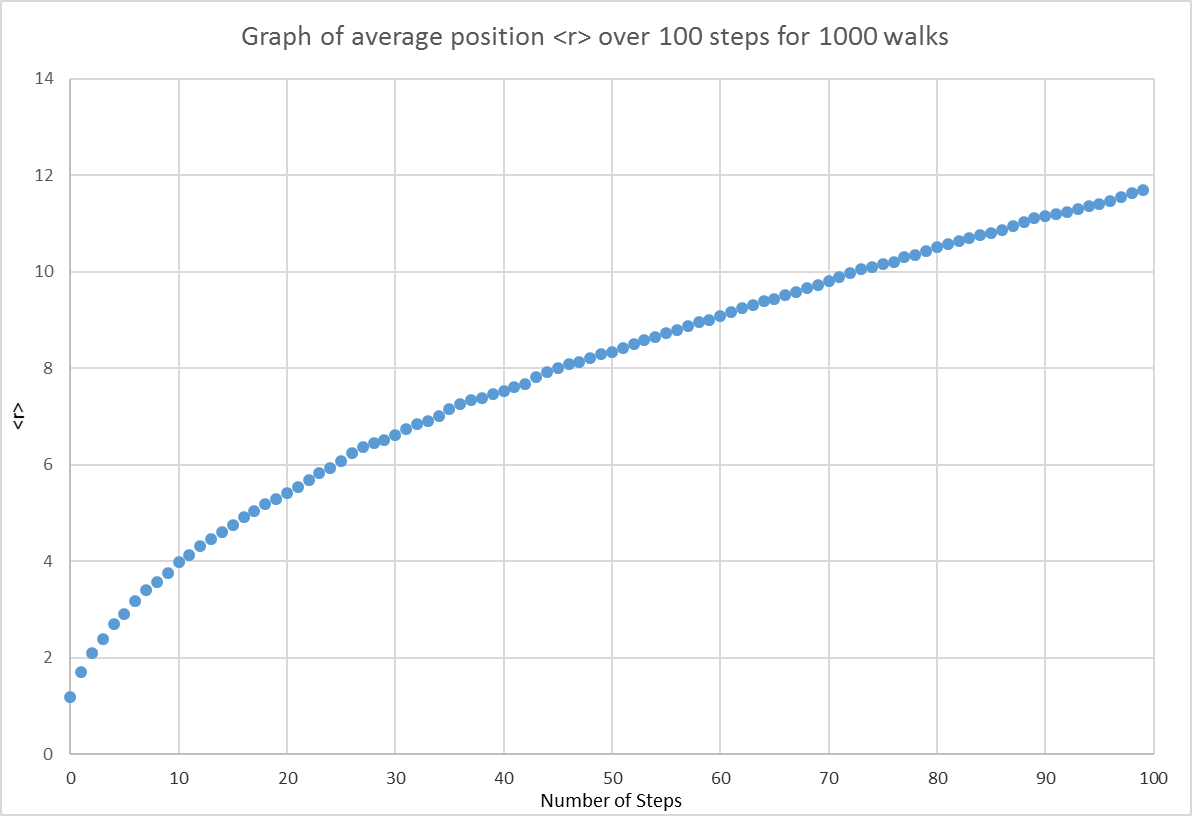
Figure 8 shows a graph of <x2> against number of steps, showing the average distance travelled at each step of the 1000 walks. This explains why the <x2> values only increase; mathematically, it is because the value is squared and cannot be negative, physically it is because distance travelled can only increase. Note that the gradient here is 0.5468, if there was an equal chance of movement and no chance of non-movement then the gradient would be close to 1 as every *p*-value induces a movement. This can be investigated further by changing the probabilities of each movement and observing the new gradient of the graph. A smaller probability of non-movement will increase the gradient of the graph and a larger probability will decrease the gradient (this will also have the effect of a higher value for <x2> being reached within 100 steps).

If the ordering of the probabilities is changed (discussed earlier) the results of the first graph will be different as a different set of movements will be made. However, as long as the probabilities are kept of the same proportions the gradient of this graph will stay the same.

Task 4: Three-dimensional random walk

The program from Task 3 has been adapted to generate a three-dimensional walk in x, y and z directions. Each direction must have independent movement. This means that when deciding if there will be a movement in each direction a separate random number will need to be generated for x, y and z for each step. Another adaption that needs to be made is to use <*r>* instead of <*x>* where *r* is the distance to the origin (0, 0, 0). The probabilities of each movement are the same, and the same RNG will be used, but it will generate the next number in the sequence and thus an independent chance of movement for that dimension. In general, the portion of the program from Task 3, which deals with the movement for x, can be repeated for *y* and *z* with the variables changed and ensuring that at the start of each walk the values are reset.

Figure 9: The average position of each step of 1000 walks – an average walk in 3-dimensions



When comparing Figure 9 to Figure 7 (comparing the random walks between 1-D and 3-D) there is a stark difference in the data. The behaviour of Figure 9 shows an average walk as the position in 3D for each step. This is found by the equation r = (x2+y2+z2)0.5, which mathematically removes any negatives from the equation, it is clear to see that this means the graph only increases with each step. This can be difficult to visualise in three dimensions but thinking through the problem shows physically why this happens. In three dimensions a negative value cannot denote an opposite direction like it can in one dimension. It can also be said that in three dimensions any one step can cause a net movement that either does not move at all or moves overwhelmingly in one direction, meaning the walk will not oscillate around zero. A three dimensional walk is said to be transient, that is, it will not return to the state that it started in, emphasised by the graph increasing and never decreasing with steps (Mare, 2013).

Figure 10: The average position squared of each step of 1000 walks – in 3-dimensinos

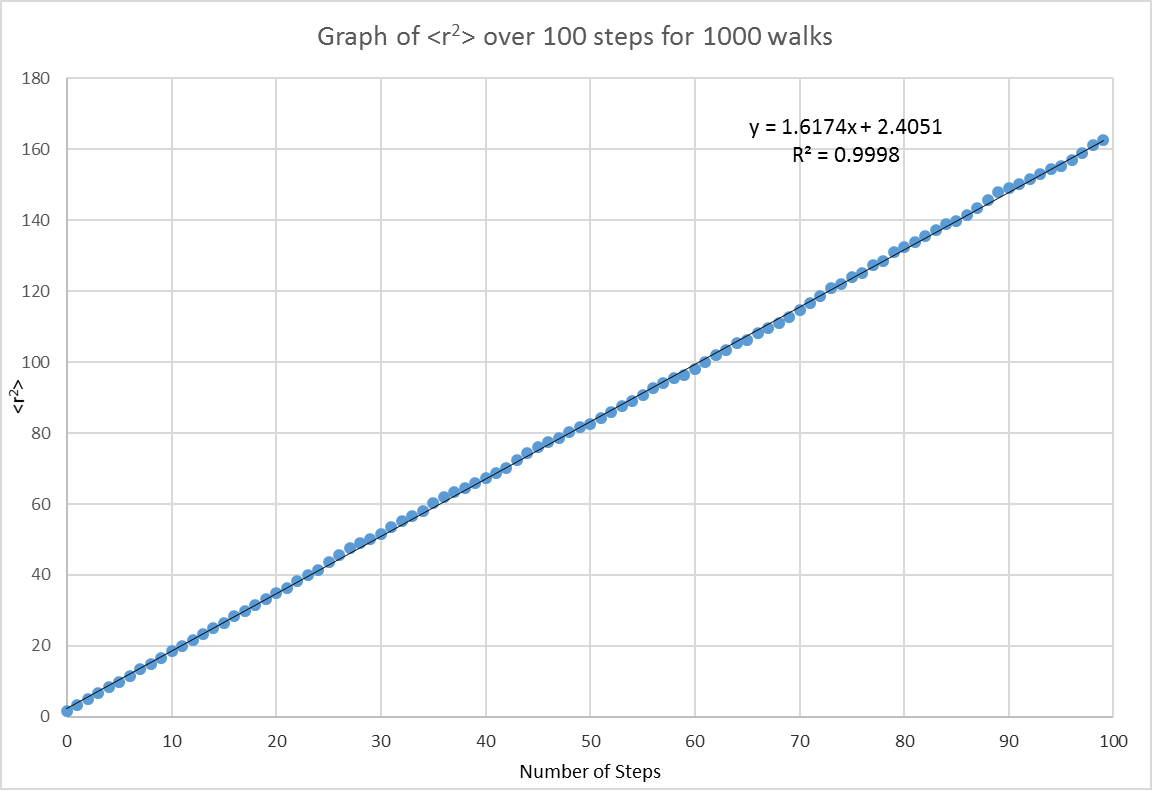


Figure 10 shows the <r2> values and how they increase with number of steps. This is a measure of the distance travelled for each step. As with Figure 8, the value is squared so it is only ever increasing. However, comparing the gradients (m(<x2>) = 0.5468; m(<r2>) = 1.6174) it can be seen that the distance in 3-D increases more rapidly than for 1-D. In fact, the gradient (and thus rate of change) is around 3 times larger for the r-values than the x-values. This may be due to movement being available in three different dimensions, with the probabilities being the same it makes sense that the rate of change is three times higher.

Further investigation shows that the gradients of Figures 8 and 10 are only affected by the probabilities of the movements, and independent of the initial seed.

Web-form - Assignment 3

Please enter your anonymous code (with no spaces) \*

83694040



Your answer

Part 1 - simple random number generator

P1: Number of iterations in cycle?

127



Your answer

Values obtained in cycle starting with n0=11

P1: n1

136



Your answer

P1: n2

79



Your answer

P1: n3

148



Your answer

P1: n4

51



Your answer

Part 2 - improved random number generator

P2: Number of iterations in cycle?

65535



Your answer

Values obtained in cycle starting with n0=last four digits of student number (e.g. 5678 for student number 12345678)

P2: n0 (last 4 digits of student number)

5427



Your answer

P2: n1

58304



Your answer

P2: n2

41665



Your answer

P2: n3

15622



Your answer

P2: n4

11423



Your answer

Part 3 - 1D random walk - <x> and <x^2> graphs

Enter below either the gradient you measured or 'NA' (without quote marks) if not appropriate to measure gradient

<x> plot, gradient (enter 'NA' if not appropriate to measure)

NA



Your answer

<x^2> plot, gradient (enter 'NA' if not appropriate to measure)

0.5468



Your answer

Part 4 - 3D random walk - <r> and <r^2> graphs

Enter below either the gradient you measured or 'NA' (without quote marks) if not appropriate to measure gradient

<r> plot, gradient (enter 'NA' if not appropriate to measure)

NA



Your answer

<r^2> plot, gradient (enter 'NA' if not appropriate to measure)

1.6174



Your answer

Appendices:

Part 1: Investigation of a Simple Random Number Generator

**#include** <stdio.h>

**#include** <stdlib.h>

**int** **main**(**void**) {

**setvbuf**(stdout,NULL,\_IONBF,0);

**setvbuf**(stderr,NULL,\_IONBF,0);

/\*setting variables dependant on student number\*/

**int** a=255-(2\*54), b=(2\*27)+1, c=256;

**int** counter=0;

**int** seedn=11;

/\*using a do…while loop so at least 1 iteration happens before condition is assessed. This means that the conditions to make the loop stop are not fulfilled before the loop has time to run once.\*/

**do**{

**printf**("%d %d\n", counter, seedn);

/\*Generating a random number using the given formula\*/

seedn=(a\*seedn+b)%(c);

/\*Using a counter to keep track of number of iterations\*/

counter++;

/\*this while condition will only allow the loop to start again if the seed not not equal the initial value. Meaning that the program will stop calculating when the formula begins repeating values\*/

}**while**(seedn!=11);

/\*printing out a final value to display that what iteration the loop finally restarts at.\*/

**printf**("%d %d", counter, seedn);

**return** EXIT\_SUCCESS;

}

Part 2: A More Robust Random Number Generator

**#include** <stdio.h>

**#include** <stdlib.h>

/\*this program is identical to part 1 except for a change in variables\*/

**int** **main**(**void**) {

**setvbuf**(stdout,NULL,\_IONBF,0);

**setvbuf**(stderr,NULL,\_IONBF,0);

**int** a=8005, b=1, c=65536;

**int** counter=0;

**int** seedn=5427;

**do**{

**printf**("%d %d\n", counter, seedn);

seedn=(a\*seedn+b)%(c);

counter++;

}**while**(seedn!=5427);

**printf**("%d %d", counter, seedn);

**return** EXIT\_SUCCESS;

}

Part 3: One Dimensional Random Walk

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <math.h>

**int** **main**(**void**) {

**setvbuf**(stdout,NULL,\_IONBF,0);

**setvbuf**(stderr,NULL,\_IONBF,0);

//setting up a file pointer to write the values to

FILE \* file\_pointer;

file\_pointer = **fopen**("A3 Part 3 Testing.xls", "w");

//variables

**int** a=8005, b=1, c=65536, seedn=5427, x;

//counters

**int** step=0, walk=0, counter=0;

/\*arrays of 100 values (0 through to 99). The {0} ensure that each slot of each array will initially be zero so that no random numbers are inserted by the program and then affect the sums.\*/

**double** sumx[100]={0}, sumsqx[100]={0};

/\*final values\*/

**double** finx, finsqx, p;

/\*initial for loop to iterate the nested code 1000 times, representing each walk\*/

**for**(walk=0; walk<1000; walk++){

/\*setting the x variable to zero at the beginning of each walk, effectively ‘resetting’ the value so that when a new walk starts it is not influenced by previous walks\*/

x=0;

/\*this nested loop is for the 100 steps taken for each walk\*/

**for**(step=0; step<100; step++){

//generating a random number

seedn=(a\*seedn+b)%(c);

//using random number seed to create a number between 0 and 1

p=(**double**)seedn/((**double**)c-1);

/\*a set of if/else statements using calculated probabilities to decide if movement is needed and in which direction. Originally a statement for doing nothing was inserted here but it was creating a break which then meant no addition was being made to the array, this was removed as nothing was need to make the program do nothing.\*/

**if**(p>0.72865){

x++;

}

**else** **if**(p<0.27135){

x--;

}

/\*summing the x values produced for each step. Note that the step variable is used again here to assign x to an array slot dependant on the step number.\*/

sumx[step]+=(**double**)x;

sumsqx[step]+=(**double**)x\*(**double**)x;

}

}

/\*once all the walks have been completed averages are needed to be taken. These must be taken outside of the main walk loop. Each array slot is divided by 1000. Note how although a different variable is used within the array it will still produce the same numbers (sumx[0], sumx[1] etc.)\*/

**for**(counter=0; counter<100; counter++){

finx=sumx[counter]/1000;

finsqx=sumsqx[counter]/1000;

//printing the results to file

**printf**("%d %g %g\n", counter, finx, finsqx);

**fprintf**(file\_pointer, "%d %g %g\n", counter, finx, finsqx);

}

**return** EXIT\_SUCCESS;

}

Part 4: Three Dimensional Random Walk

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <math.h>

**int** **main**(**void**) {

**setvbuf**(stdout,NULL,\_IONBF,0);

**setvbuf**(stderr,NULL,\_IONBF,0);

/\*setting up file pointers and where to write data to\*/

FILE \* file\_pointer;

file\_pointer = **fopen**("A3 Part 4 Testing.xls", "w");

**int** a=8005, b=1, c=65536, seedn=5427, x, y, z;

**int** step=0, walk=0, counter=0;

**double** sumr[100]={0}, sumsqr[100]={0};

**double** finr, finsqr, p;

//for loop to generate each walk

**for**(walk=0; walk<1000; walk++){

//resetting each variable at the start of each walk

x=0;

y=0;

z=0;

//for loop to iterate over each step

**for**(step=0; step<100; step++){

/\*generating random number for x values and converting to a value between 0 and 1\*/

seedn=(a\*seedn+b)%(c);

p=(**double**)seedn/((**double**)c-1);

//using random number to decide on movement for x values

**if**(p>0.72865){

x++;

}

**else** **if**(p<0.27135){

x--;

}

/\*generating new random value for y values and converting to a value between 0 and 1\*/

seedn=(a\*seedn+b)%(c);

p=(**double**)seedn/((**double**)c-1);

//using random number to decide on movement for y values

**if**(p>0.72865){

y++;

}

**else** **if**(p<0.27135){

y--;

}

/\*generating random number for z values and converting to a value between 0 and 1\*/

seedn=(a\*seedn+b)%(c);

p=(**double**)seedn/((**double**)c-1);

// using random number to decide on movement for z values

**if**(p>0.72865){

z++;

}

**else** **if**(p<0.27135){

z--;

}

/\*calculating the r values and summing them in an array for r. the same is being done for the r squared values\*/ sumr[step]+=**pow**(**pow**((**double**)x,2)+**pow**((**double**)y,2)+**pow**((**double**)z,2),0.5); sumsqr[step]+=**pow**(**pow**((**double**)x,2)+**pow**((**double**)y,2)+**pow**((**double**)z,2),0.5)\***pow**(**pow**((**double**)x,2)+**pow**((**double**)y,2)+**pow**((**double**)z,2),0.5);

}

}

/\*for loop to find the average of each step for the r values and r squared values and printing these values to the file defined above\*/

**for**(counter=0; counter<100; counter++){

finr=sumr[counter]/1000;

finsqr=sumsqr[counter]/1000;

**printf**("%d %g %g\n", counter, finr, finsqr);

**fprintf**(file\_pointer, "%d %g %g\n", counter, finr, finsqr);

}

**return** EXIT\_SUCCESS;

}

References

Kaye, B. H. (1994) *A random walk through fractal dimensions*, 2nd edn., Weinheim: VCH Verlagsgesellschaft.

L’Ecuyer, P. (2017) *History of uniform random number generation,* Available at: *https://www.iro.umontreal.ca/~lecuyer/myftp/papers/wsc17rng-history.pdf* (Accessed: 2nd December 2018).

Mare, S. (2013) *Polya’s recurrence theorem,*Available at: *https://math.dartmouth.edu/~pw/math100w13/mare.pdf* (Accessed: 30th November 2018).