Random Numbers and Monte Carlo Simulations

Mitchell Miller

Physics 240

This assignment explores the applications and methods of random generators. Random number generators can be used to simulate all kinds of processes in nature when one cannot possibly account for or does not know every force or field that impacts the result. This lab explores a few different applications of random simulations. First, the random walk problem is explored. Next, radioactive decay is explored. Finally, integration using Monte Carlo methods is explored. The Monte Carlo method of integration works by picking random points and checking whether or not they are in the area being integrated. In addition to these applications, two methods of random generation were examined. The first is the linear congruent method. This method works by manipulated a starting value with a simple algorithm that generates a repeating list of pseudo-random numbers.

1. Introduction

Random number generators are incredibly valuable in scientific modeling and simulations. By being able to simulate random problems, one can significantly cut down on computational power and time necessary to complete the simulation. Although computers cannot generate truly random numbers, the values that are made so diverse that finding a pattern is almost impossible. One method explored in this assignment, the linear congruent method, generates a list of numbers that repeat at a set interval and are really not random at all. This method is the simplest, but only useful when very low amounts of randomness are needed. The built-in random number generator 'drand48' is a very powerful tool, generating highly random numbers. This generator is random enough for most basic scientific simulations and more than enough for the simulations conducted in this lab. The first simulation explored was the random walk. This simulates the path of a particle moving randomly from the origin in two dimensions. This simulation can be used to model diffusion or other problems. Also, this model assumes that each step is independent and there is no pause between steps. The next simulation is based on radioactive decay. This models a number of atoms and keeps track of how many are left due to decay after a given time. Finally, the Monte Carlo method of integration is explored in this assignment. This method works essentially by throwing stones (random numbers) at a pond (area of integration) and checking whether or not they are in the area. This is considered the fastest and generally most accurate method of integration.

2. Theory

The first method of random number generation in this lab is the linear congruent method or the power residue method. This works by having the user set two constants, a starting value called the seed, and a modulus value[1]. This method generates a set of random numbers between zero and one less than the modulus value. Additionally, it generates only as many random numbers as the modulus and repeats the same series of numbers every time it returns to the seed value. The algorithm for this method is as follows[2]

$$r_{i+1} = (ar_i + c)mod(M) = remainder\left(\frac{ar_i + c}{M}\right) \quad (1)$$

Where *a* and *c* are constants and *M* is the modulus value. Using this method, if *M* were chosen to be 9, one would generate a list of random numbers between 0 and 8 that includes each number once in a pseudo-random order and repeats identically after 9 terms. It is also very simple to scale this to any range of numbers which allows it to be generalized for many problems.
The built-in random number generator, drand48, is also incredibly useful. This is a simple function that generates a random decimal between 0 and 1. Using this random number generator, one can simulate highly random processes. There is very little repetition in this function. This method also uses the linear congruent method and 48-bit integer arithmetic[3]. This function uses the following constants for the linear congruent method[4]

$$M = 2^{48}, c = 11, a = 25214903917 \quad (2)$$

This clearly generates an enormous number of distinct points with 2^{48} different numbers before repeating. Using this function for most simple simulations, one would never see repetition in the sequence.
The first simulation conducted in this lab is the random walk. This simulates thermal motion and particle collisions in a gas as well as diffusion[5]. This simulation calculates a random value for a step in the x and y direction. After a given number of steps, the 'particle' ends at a distance from the origin. The average distance is related to the number of steps as follows[6]

$$R_{rms} \cong \sqrt{N} r_{rms} \quad (3)$$

This means that, on average, the particle travels a distance approximately equal to the average step size times the square root of the number of steps.
The next simulation conducted was for radioactive decay. This simulation calculates how many radioactive particles remain of a sample after a given time. This can be calculated with the following equation

$$\frac{dN(t)}{dt} = -\lambda N(0)e^{-\lambda t} \quad (4)$$

Where λ is the decay rate. This is an exponential decay towards zero from a starting value of N(0). In application, a log plot of this generates a simple line until a point where it becomes very erratic due to the relatively small number of particles remaining. At this point, the random nature becomes very noticeable.
The final application of random number generators in this lab is in Monte Carlo integration. This method works by using random values to guess at the area of integration. The random value is tested to see whether or not it is in the desired range and a counter keeps track of how many are in and out. Using these counters one can find the value of the integral using the following

$$A_{Integral} = \frac{N_{In}}{N_{In} + N_{Out}} * A_{Total} \quad (5)$$

Using a relatively small number of trials, one can quickly calculate an approximation for the area with very simple code. This is an incredibly valuable method because of its speed and flexibility. Additionally, for high dimension integration, the methods explored in Assignment 3 require an impossibly large number of loops. This method allows one to calculate an integral in many dimensions with relative ease.

3. Experimental Method

The first part of this lab explores the linear congruent method of random number generation. Using equation (1), an algorithm to generate a series of random numbers was implemented. Using this, the quality of the sequence generated was assessed by plotting successive pairs and testing different constants and moduli.
Next, a random walk simulation was conducted. A random path was calculated by increasing or decreasing the x and y values in a set number of steps. The average distance from the origin was calculated for several trials.
Next, a radioactive decay simulation was performed. Using equation (4), the number of particles was calculated at constant time steps until no particles were left.
After that, the area of a circle was calculated using a Monte Carlo method. Using equation (5), the area, π, was calculated to two decimals.
Finally, the following simple ten dimensional integral was computed using the Monte Carlo method.

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \dots \int_0^1 dx_{10}\,(x_1 + x_2 + \cdots + x_{10})^2 \qquad (6)$$

4. Data and Analysis

The first part of this lab explored the properties of the linear congruent method. Using a poor choice of constants, the results of this method are displayed in Figure 1. Although they appear to be fairly random, Figure A shows how easily predictable they really are.

Figure A

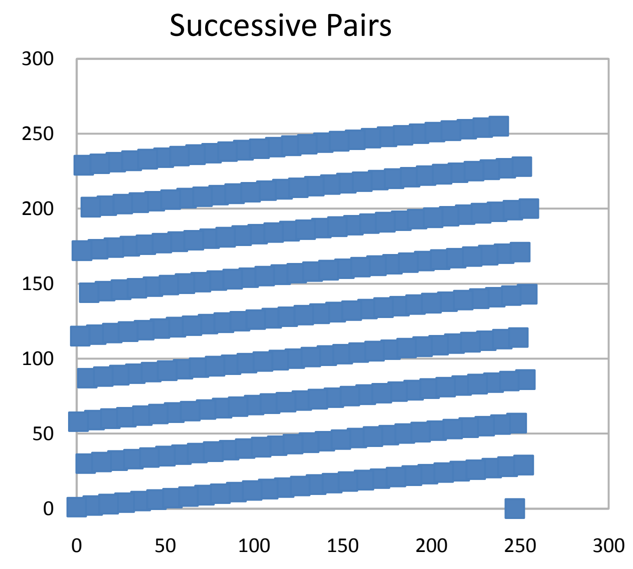

Figure A – This figure displays the successive pairs of the linear congruent method with poor constants. (x , y) = (r_i , r_{i+1})

By analyzing the sequence of numbers generated by this method, one finds that the period of the sequence is equal to the modulus value. Figure 2 shows a comparison for Figure 1. This figure shows the same number of points, however these points were generated by the built-in random number generator. Finally, the linear congruent method was used to generate a sequence using 'good' constants. The results of this are displayed in Figure 3. These values are random enough to be useful in scientific data since they only repeat every 112233 values.
Next, the random walk simulation was conducted. Figure 4 displays the path of the simulated particle for one trial conducted. It is clear from this figure that the path is random; however it is important to note that the particle never continues in any direction for more than a couple steps and it generally stays close to the origin.

Figure 5 displays the relation between the number of steps taken and the average distance from the origin the particle ends at. This figure shows that the distance traveled and the number of steps are highly related.

The next simulation, the radioactive decay, was also successfully tested. Figure 6 shows the results of this simulation. As time passes, the log of the remaining atoms versus time is linear, until it becomes erratic at around t = 35. This is due to the random decay and relatively small number of atoms remaining. There is a longer time between the last few numbers of atoms decaying. Figure 8 displays the results of several different trials with different starting amounts. This figure shows that the decay rate is independent of the starting amount. It also shows that the slope of the log of the remaining atoms versus the time is equal to the decay rate. In this simulation, the decay rate was set at 0.1, which is very close to the slopes displayed. Although this process is random, the decay is exponential. This is due to the probability distribution of decay time.

Finally, the Monte Carlo method of integration was tested. First, the area of the unit circle was calculated using equation (5). The value of π was calculated to two decimals of accuracy in less than 150,000 guesses. This may seem like a large number of trials, but in reality it is incredibly fast since all that is done is a simple guess and check. There is really no computationally intensive calculation necessary for this method. Lastly, a 10 dimensional integral was calculated. By comparing the calculated value to the analytical value of 155/6, a plot of error versus the square root of number of loops was generated in Figure 9. This shows that as one increases the number of loops, the error greatly decreases.

5. Conclusion

This lab successfully demonstrated the applications of random number generation methods. First, the linear congruent method was explored. Next, two simulations using random numbers were conducted, random walk and radioactive decy. Finally, Monte Carlo Integration was explored with a simple circle and a 10 dimensional problem.

6. Bibliography

[1]"Linear Congruential Generator." *Wikipedia, the Free Encyclopedia*. Web. 05 Apr. 2010. <http://en.wikipedia.org/wiki/Linear_congruential_generator>.

[2]"Linear Congruence Method -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 05 Apr. 2010. <http://mathworld.wolfram.com/LinearCongruenceMethod.html>.

[3]"Drand48." *The Open Group - Making Standards Work*. Web. 05 Apr. 2010. <http://opengroup.org/onlinepubs/007908799/xsh/drand48.html>.

[4]"Drand48() — Pseudo-Random Number Generator." *IBM Support & Downloads - United States*. Web. 05 Apr. 2010. <http://publib.boulder.ibm.com/infocenter/zos/v1r10/topic/com.ibm.zos.r10.bpxbd00/rdrnd4.htm>.

[5]"Random Walk -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 05 Apr. 2010. <http://mathworld.wolfram.com/RandomWalk.html>.

[6]"Random Walk--2-Dimensional -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 05 Apr. 2010. <http://mathworld.wolfram.com/RandomWalk2-Dimensional.html>.

APPENDIX A: TABLES AND FIGURES

Figure 1

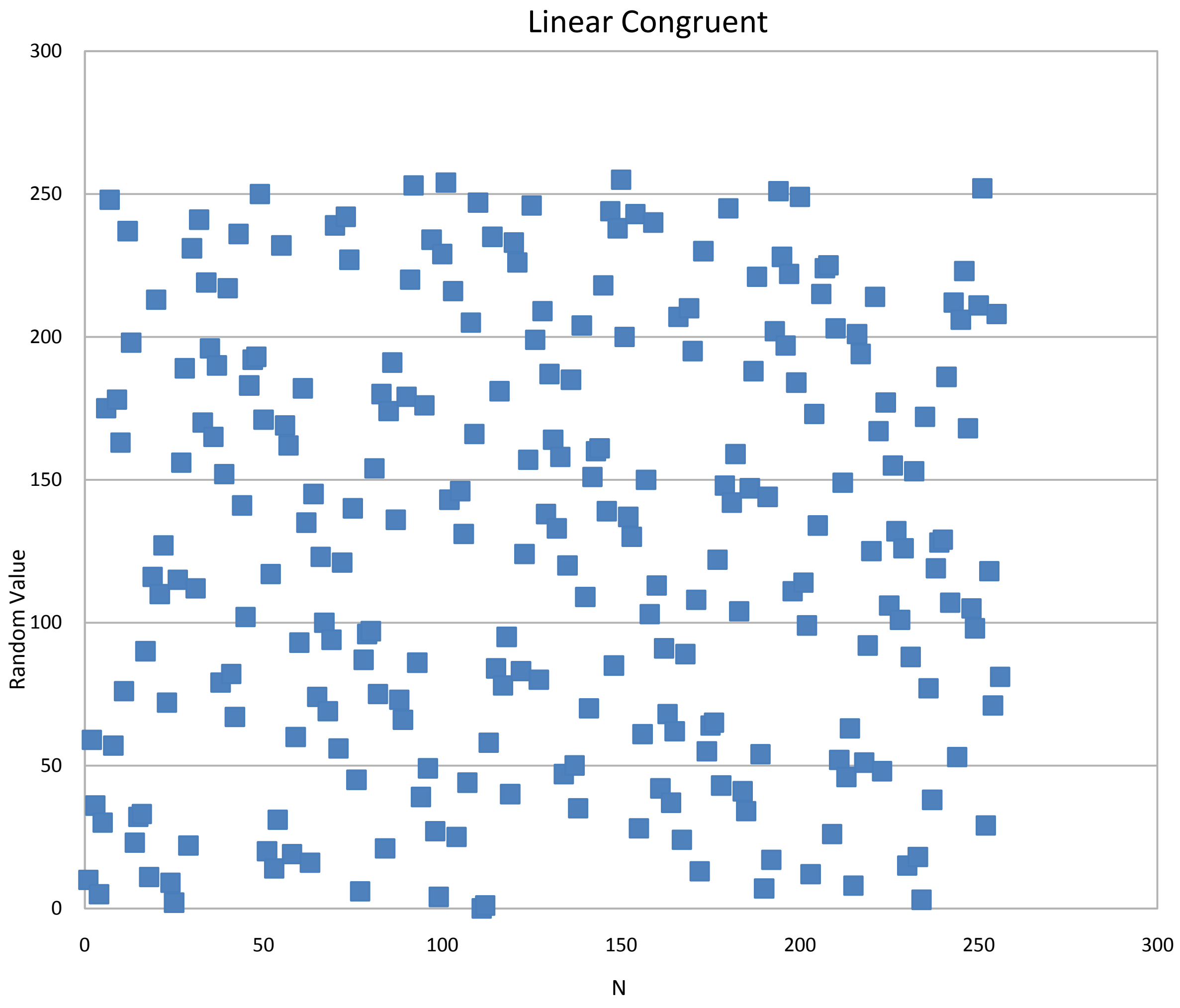

Figure 1 - This figure displays the results of the linear congruent method. The x-axis displays the position in the sequence of values and the y-axis the value of the random number. The constants for this were *a = 57, c = 1, M = 256,* R_1 *= 10*.

Figure 2

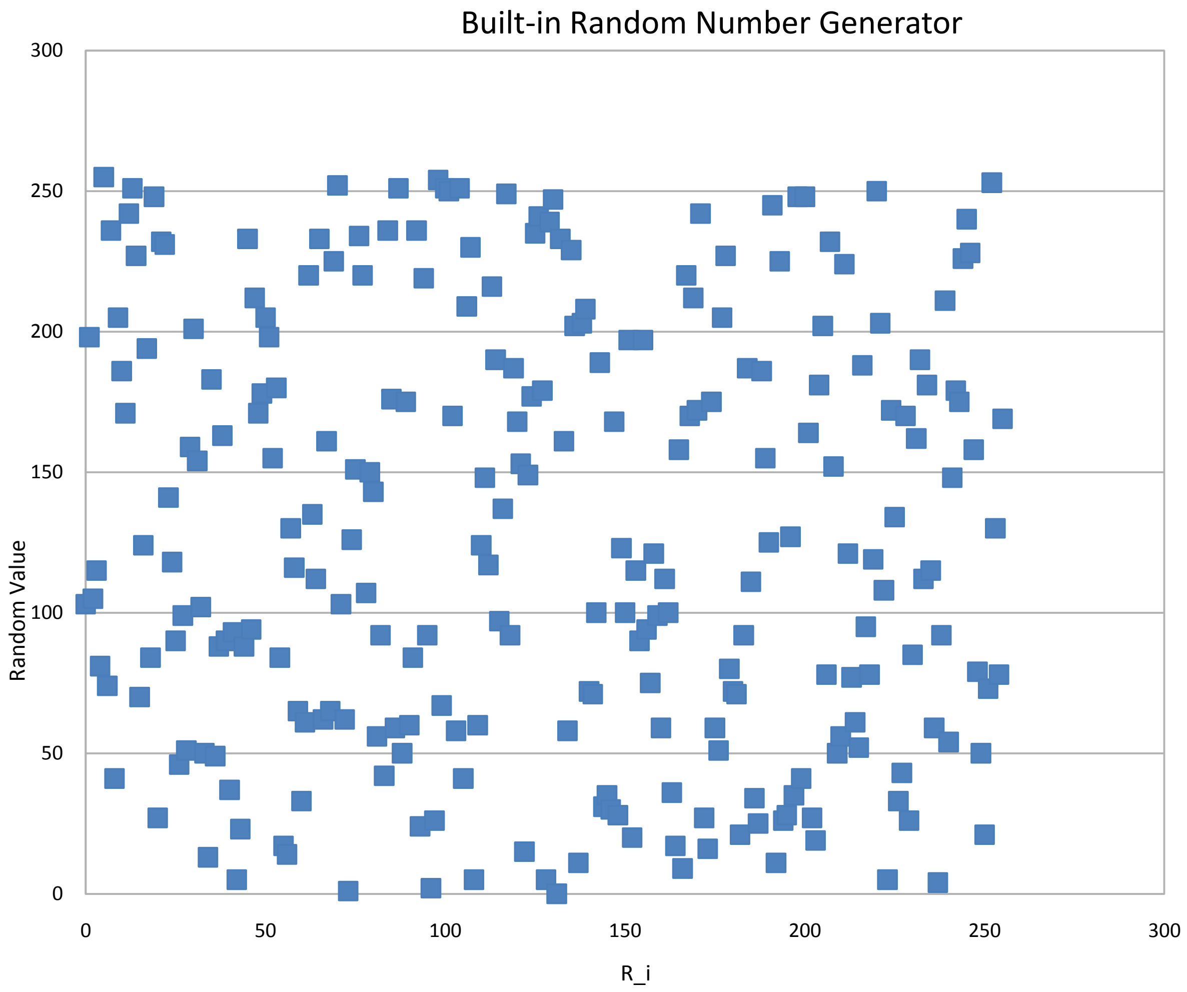

Figure 2 – This figure displays a series of random numbers generated by the function ‘drand48()’. The x-axis displays the position in the sequence of values and the y-axis the value of the random number.

Figure 3

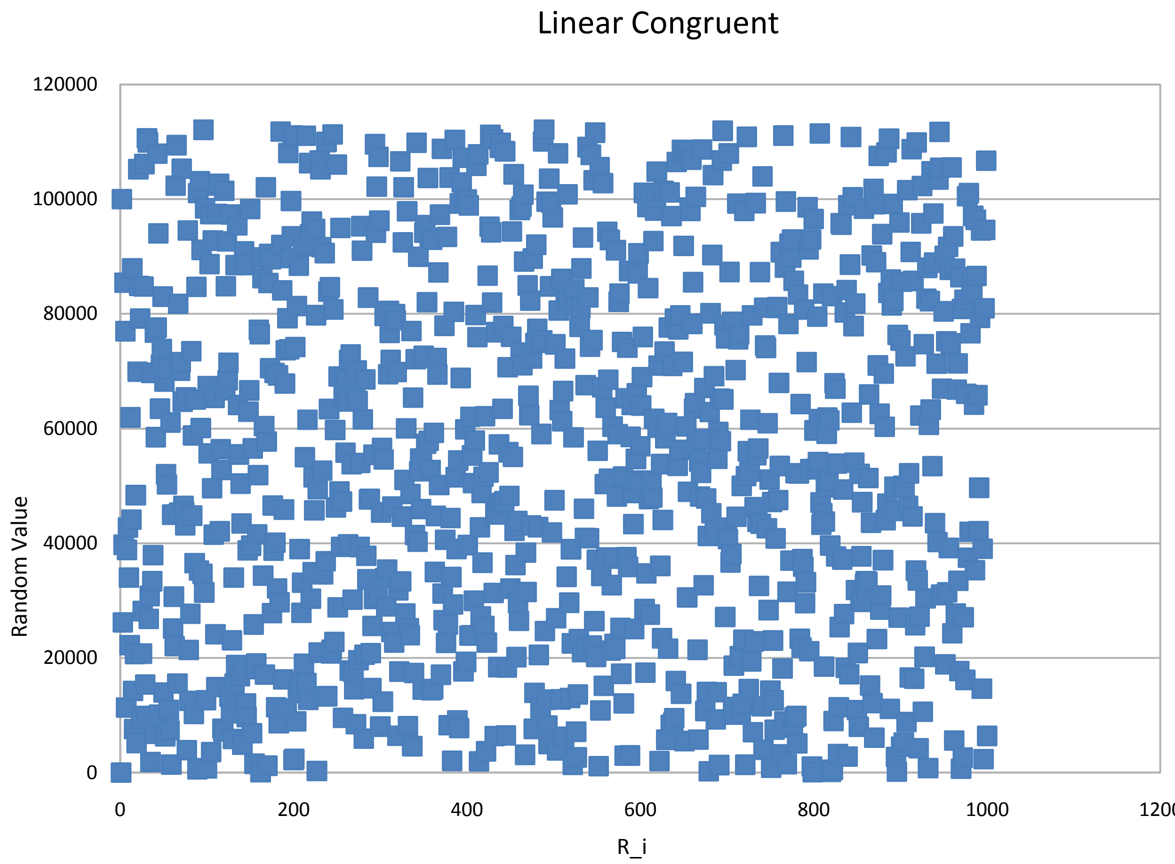

Figure 3 – This figure displays values generated using the linear congruent method with good constants. The x-axis displays the position in the sequence of values and the y-axis the value of the random number. The constants for this were $a = 9999$, $c = 11$, $M = 112233$, $R_1 = 10$.

Figure 4

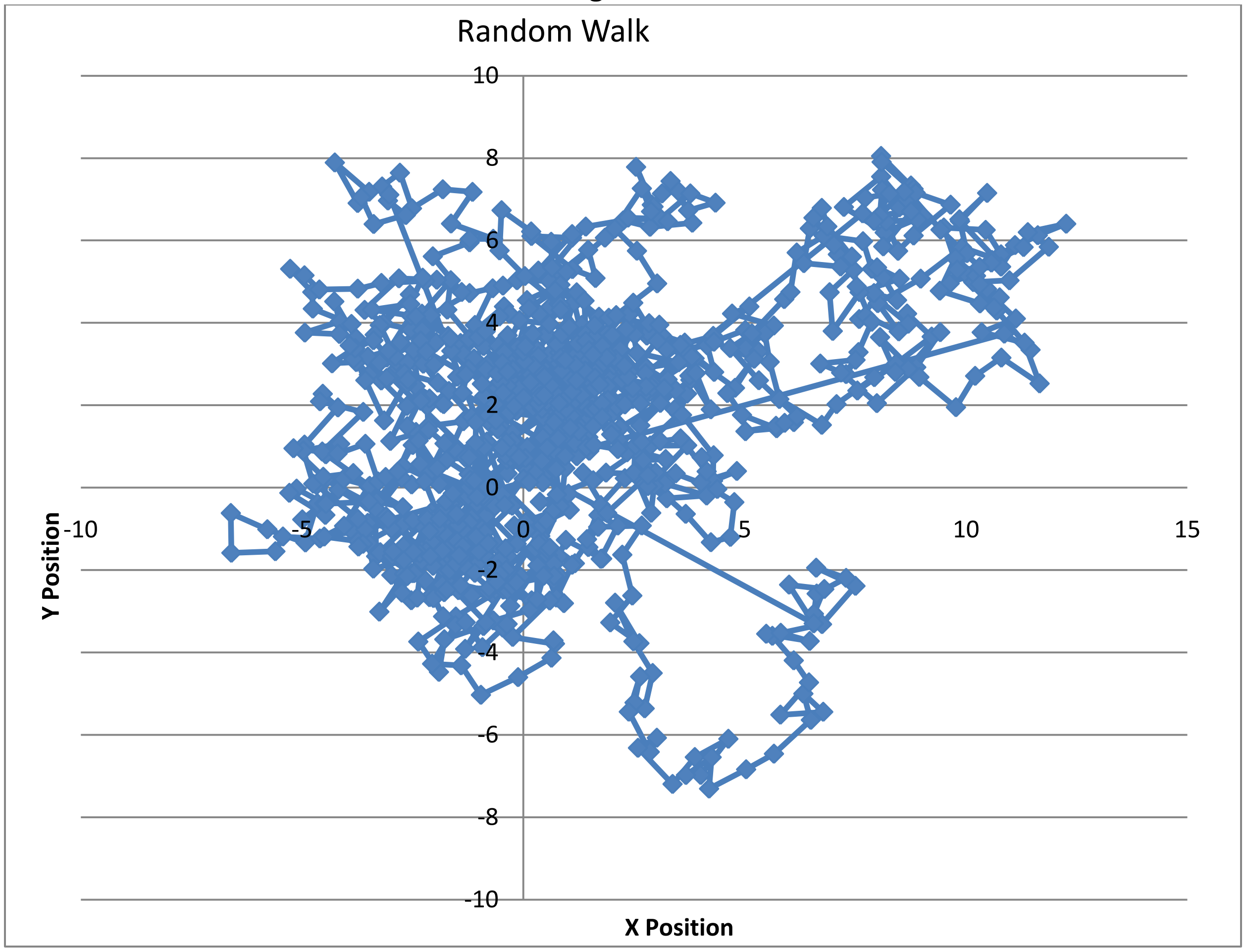

Figure 4 – This figure displays the path of a particle during random walk.

Figure 5

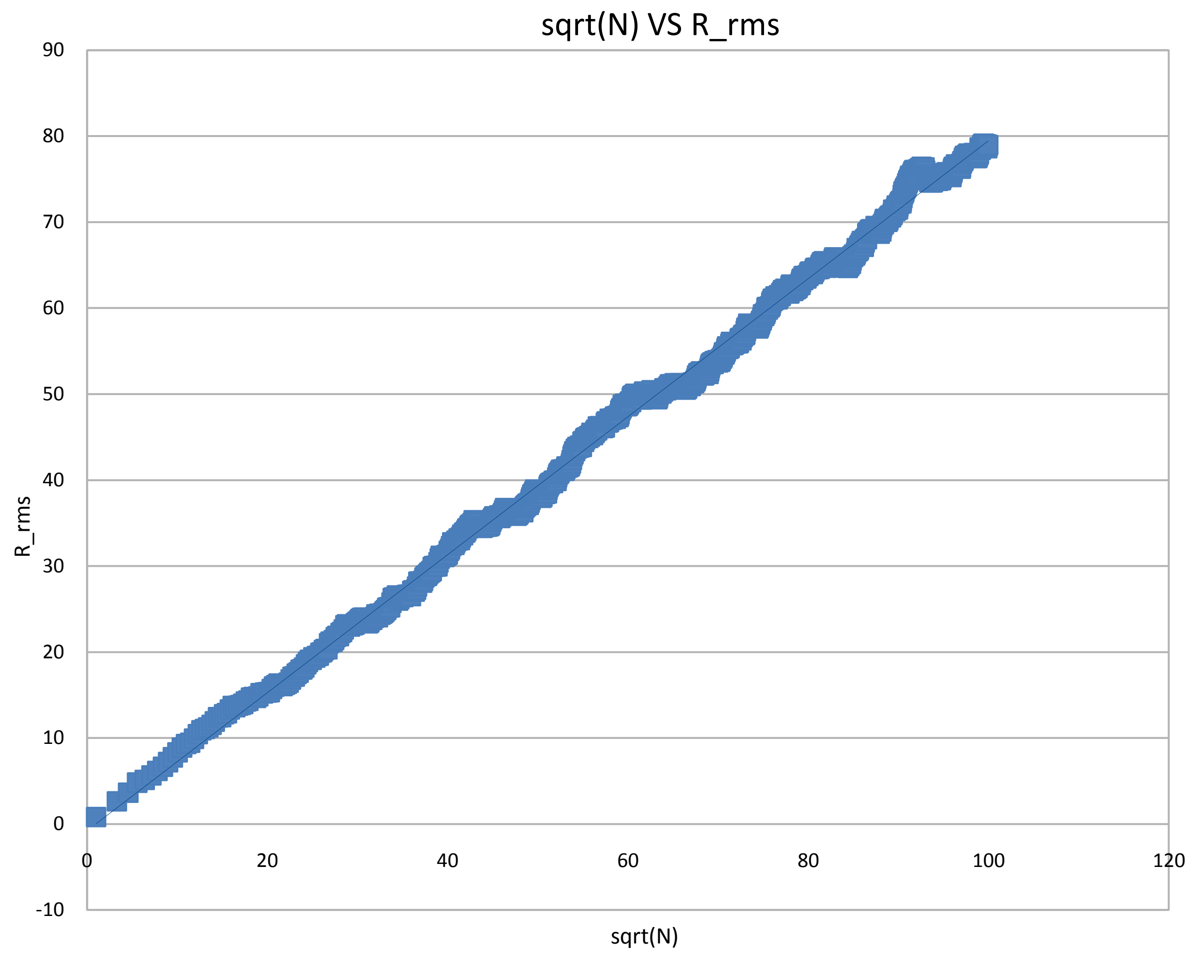

Figure 5 – This figure displays the plot of the RMS distance traveled from the origin versus the square root of the number of steps. This is linear, as expected.

Figure 6

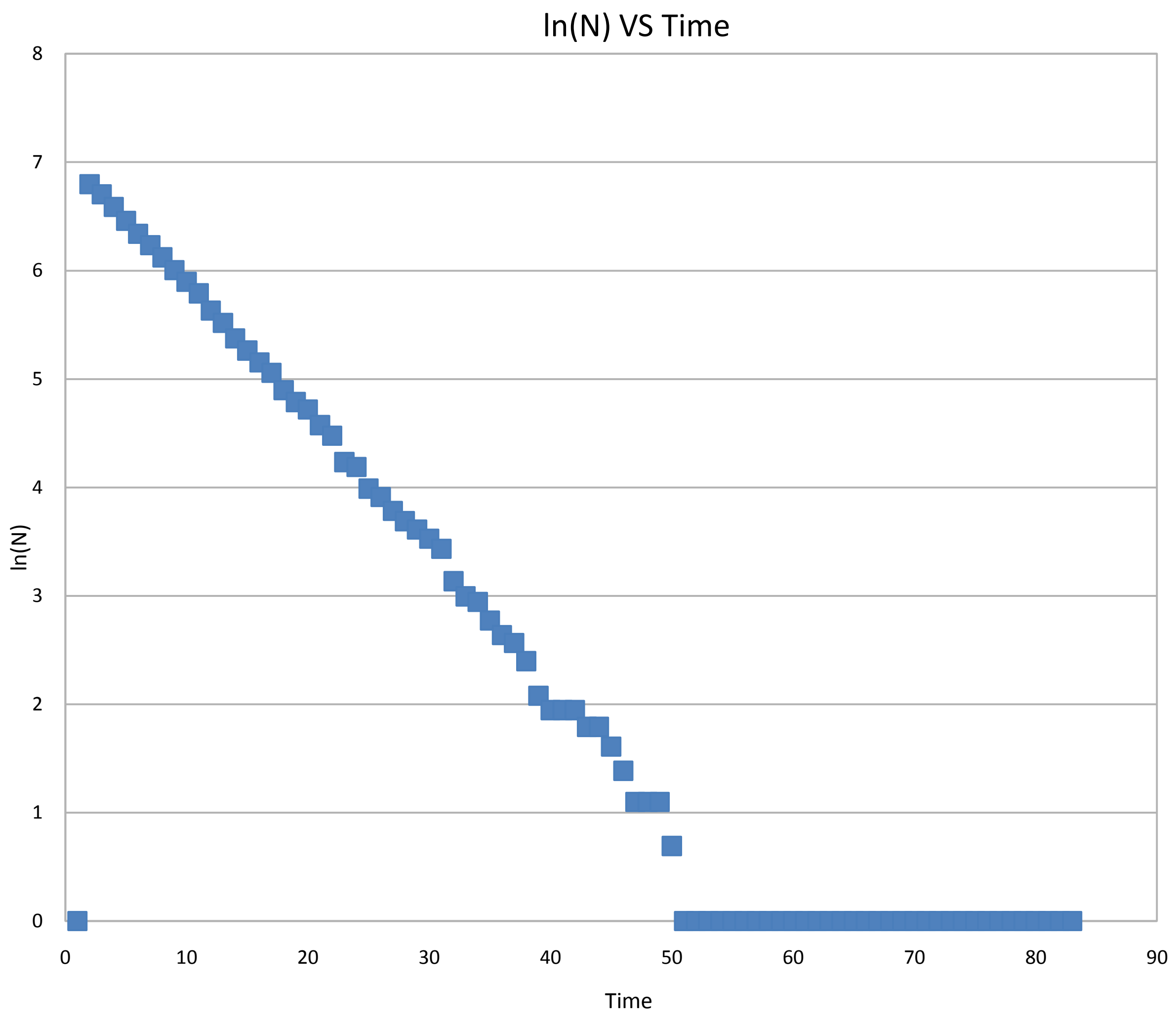

Figure 6 – This figure displays the results of the radioactive simulation. The natural log of remaining atoms is plotted against the time that has passed. This is linear, indicating an exponential decay.

Figure 7

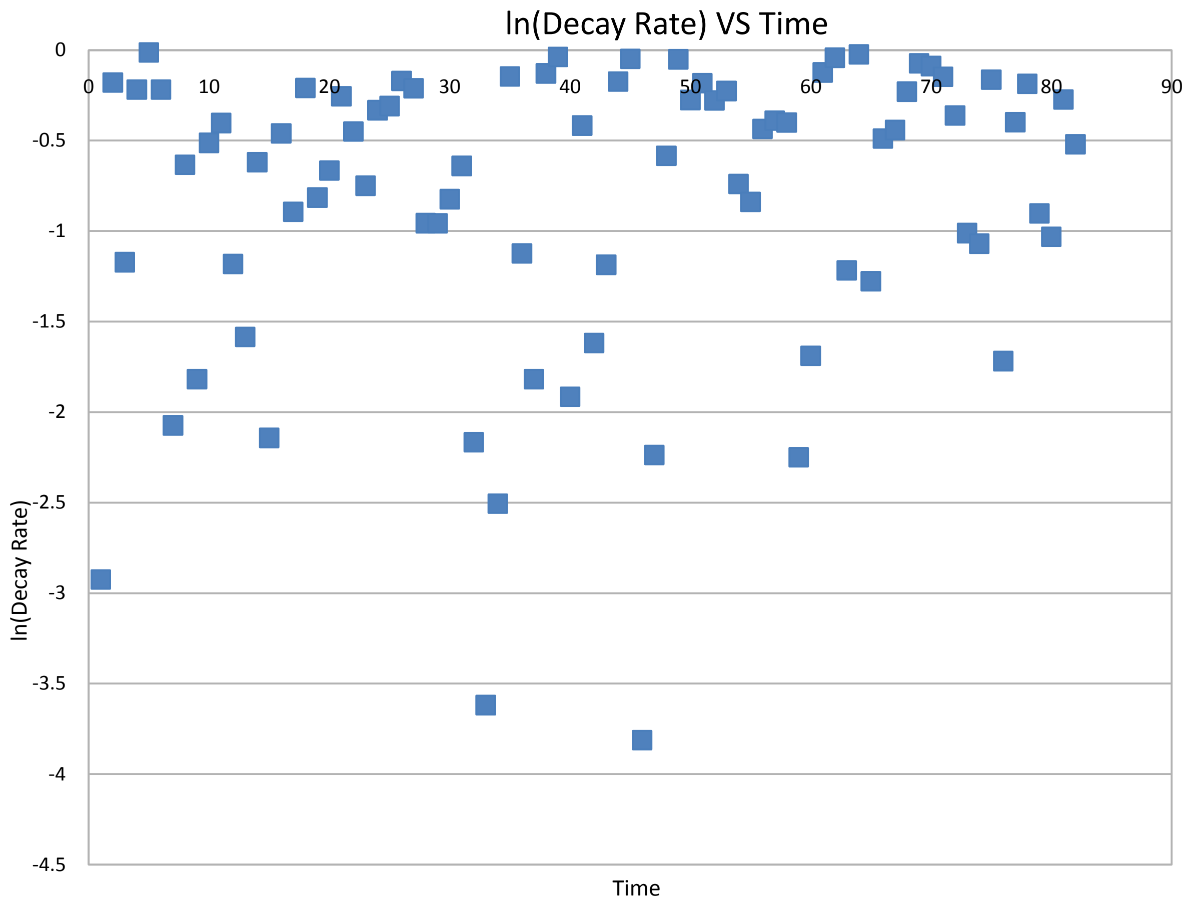

Figure 7 – This figure shows the log of the decay rate versus time. This is a random function as shown.

Figure 8

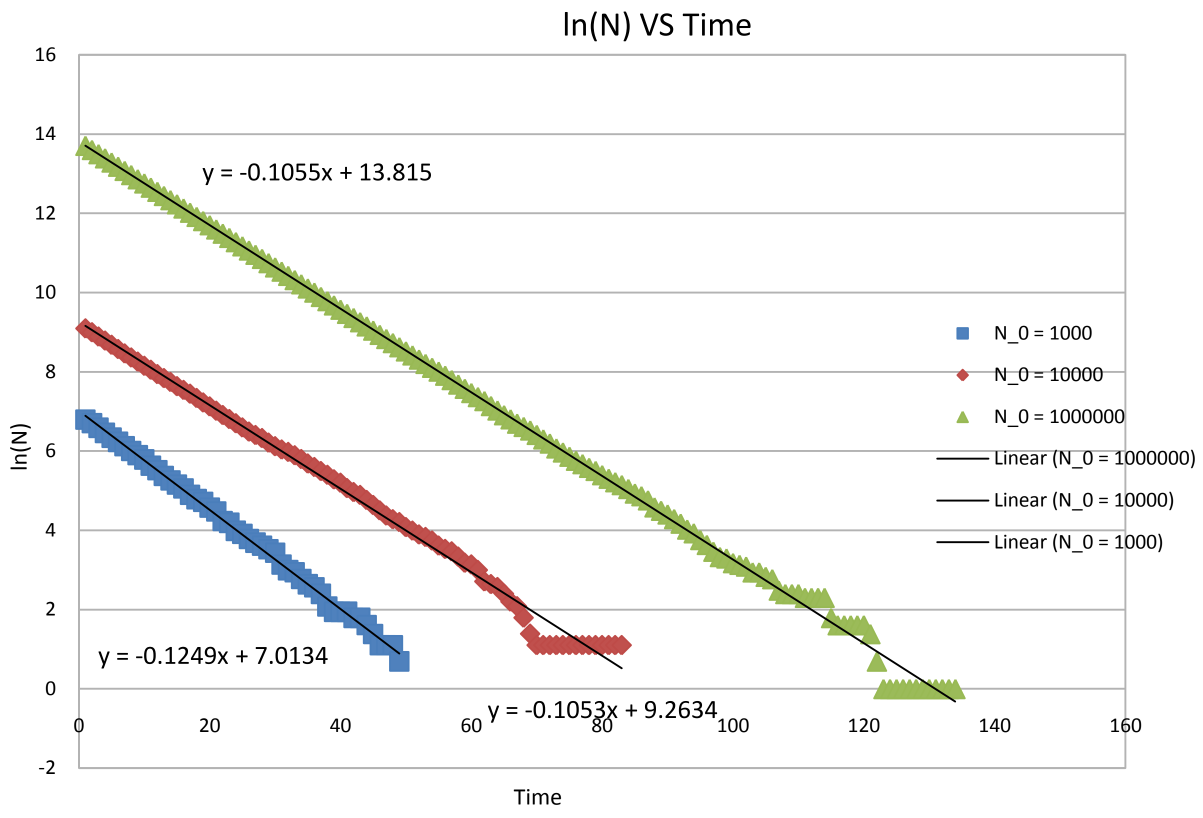

Figure 8 – This figure displays the results of radioactive decay for multiple starting values of atoms. It is clear that the average decay rate is the same regardless of starting amount. Additionally, the slope is approximately equal to the decay rate.

Figure 9

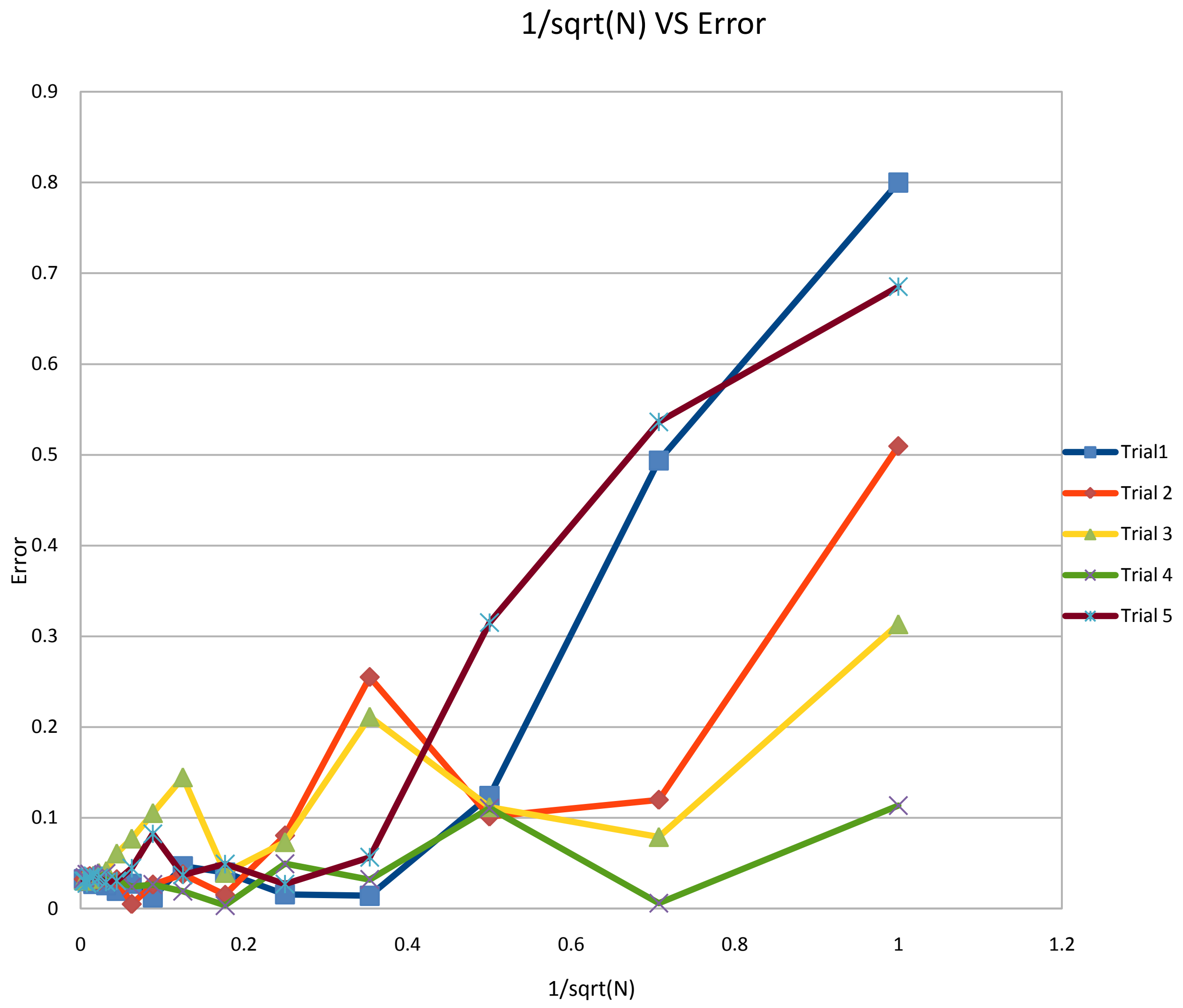

Figure 9 – This figure shows the error versus the square root of the number of loops in a 10 dimensional Monte Carlo integral. As the number of loops conducted increases, the error rapidly decreases. This makes the Monte Carlo method very fast and accurate.

APPENDIX B: SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 5
//PHYS 240 4/2/10

int linearCongruent(int constant1, int constant2, int modulus, long seed){

	//Initialize variables
	int randomCurrent, randomPrevious=seed;
	int *list;
	int counter;
	list = calloc(modulus,sizeof(int));
	list[0]=seed;
	FILE *outfile;
	outfile = fopen("5.2.2.txt","w");

	//Calculate random values
	for(counter=1;counter<=modulus;counter++){

		randomCurrent = (constant1*randomPrevious+constant2) % modulus;
		list[counter] = randomCurrent;
		//fprintf(outfile,"%d\t%d\n",randomPrevious,randomCurrent);
		randomPrevious = randomCurrent;

	}

	for(counter=1;counter<=modulus;counter++){

	printf("Random # %d = %d\n",counter,list[counter-1]);
	fprintf(outfile,"%d\t%d\n",counter,list[counter-1]);

	}

}

int main(){
	//Intialize variables
	int *list,counter;
	FILE *outfile;
	outfile = fopen("5.2.2.rand.txt","w");
	list = calloc(112233,sizeof(int));

	//Calculate random values using built in function
	for(counter=0;counter<112233;counter++){

		list[counter] = rand() % 112233;
		fprintf(outfile,"%d\t%d\n",counter,list[counter]);

	}

	linearCongruent(9999,11,112233,10);

}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 5
//PHYS 240 4/2/10

int main(){

	//Intialize Variables
	double dx,dy,x=0,y=0,meanSquare=0;
	double dxAverage,dyAverage,Average,trialAverage=0;
	int counter1,counter2,counter3;
	FILE *outfile;
	outfile = fopen("5.4.2.6.txt","w");
	//outfile = fopen("5.4.2.txt","w");
	//fprintf(outfile,"Run 1\n");

	for(counter3=1;counter3<10000;counter3+=10){

	meanSquare = 0;

		for(counter1=0;counter1<100;counter1++){

			//Set seed and start value
			srand48(495236*cos(counter1));
			x=0;
			y=0;
```

```
                    dxAverage=0;
                    dyAverage=0;

                    for(counter2=0;counter2<counter3;counter2++){

                              dx = (drand48() - 0.5) * 2;
                              dy = (drand48() - 0.5) * 2;
                              x = x + dx;
                              y = y + dy;
                              dxAverage = dxAverage + dx * 1/10000;
                              dyAverage = dyAverage + dy * 1/10000;
                              //fprintf(outfile, "%lf\t%lf\n",x,y);

                    }

                    //Compute Average values for R^2 and (dx*dy)/R^2
                    Average = dxAverage * dyAverage / (x*x + y*y);
                    trialAverage = trialAverage + Average * 0.01;
                    meanSquare = meanSquare + (x*x + y*y) * 0.01;
                    //printf("(dx*dy)/R^2 = %e\n",Average);

                    //fprintf(outfile,"\n\n\nRun %d\n",counter1+2);

          }

     printf("Mean Square Distance = %lf\n",meanSquare);
     printf("<(dx*dy)/R^2> = %e\n",trialAverage);
     fprintf(outfile, "%lf\t%lf\n",sqrt(counter3),sqrt(meanSquare));

     }

     fclose(outfile);

}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//Mitchell Miller
//Assignment 5
//PHYS 240 4/2/10

int main(){
	//Intialize variables
	double lambda = 0.1;
	double decay;
	int time, numberOfAtoms=1000, counter, dummyVariable=1000;
	FILE *outfile;
	outfile = fopen("5.5.3.txt","w");
	srand48(68111);				//Set seed value
	fprintf(outfile,"0\t1\t%.20lf\t%.20lf\n",log(1000),log(1));

	for(time=1;time<=500;time++){

		//Decay loop
		for(counter=1;counter<=numberOfAtoms;counter++){

			//Set random decay value
			decay = drand48();
			if(decay<lambda) dummyVariable--;  //Decreases number of particles

		}

		numberOfAtoms = dummyVariable;
		fprintf(outfile, "%d\t%.20lf\t%.20lf\t%.20lf\t%.20lf\n",time,
(double)numberOfAtoms/1000,log((double)numberOfAtoms),log((double)numberOfAtoms/100
0),decay);

	}

	fclose(outfile);

}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159265358979323846264338 32

//Mitchell Miller
//Assignment 5
//PHYS 240 4/2/10

int main(){

	//Intialize variables
```

```
        double randomCurrent,randomPrevious,area,location;
        int counter, nIn=0, nOut=0;
        srand48(213548);
        randomPrevious = (drand48() - 0.5) * 2;

        for(counter=0;counter<150000;counter++){

                //Calculate first random number
                randomCurrent = (drand48() - 0.5) * 2;
                location = randomCurrent*randomCurrent + randomPrevious*randomPrevious;

                //Test if point is in circle
                if(location<1)  nIn++;
                else            nOut++;
                randomPrevious = randomCurrent;

        }

        area = 4 * (double)(nIn) / ((double)nIn + (double)nOut);
        printf("Area = %lf\tnIn = %d\n", area, nIn);
        printf("Error = %lf\n", (area-PI)/PI);

}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159265358979323846264338327

//Mitchell Miller
//Assignment 5
//PHYS 240 4/2/10

int main(){

        //Intialize variables
        double x,y,average=0,counter=1;
        int counter1,counter2,counter3;
        FILE *outfile;
        outfile = fopen("6.6.3.txt","w");

        for(counter1=0;counter1<16;counter1++){

                //Reset variables
```

```
		y = 0;
		counter2 = 0;
		counter3 = 0;
		counter = 0;

		printf("********* Trial # %d *********\n",counter1+1);

		for(counter2=1;counter2<70000;counter2++){

			//Reset X
			x = 0;

			//Add to X
			for(counter3=0;counter3<10;counter3++)     x += drand48();

			//Calculate Y
			y += x * x;

			if(counter2%(int)pow(2,counter) == 0){

				counter++;
				printf("Loops = %d\tIntegral = %lf\tError = 
%lf\n",counter2,y/counter2,fabs((y/counter2)-155/6)/(155/6));

	fprintf(outfile,"%lf\t%lf\n",1/sqrt((double)counter2),fabs((y/counter2)-155/6)/(155/6));
			}

		}

		average += y/counter2;

	}

	printf("Average = %lf\n", average/16);

}
```