Math 466/666: Programming Project 1

1. This question explores why the mantissas of the floating-point numbers which appear in typical computations follow the reciprocal distribution.

(i) Consider the primary-school multiplication table which shows all products for the numbers 1 through 9. Make a bar-chart depicting how many of the 81 products in that table begin with the digit 1, how many with a 2 and so forth up to 9.

I wrote a program called p1i.c which looks like

```c
#include <stdio.h>

int count[10];
int leading(int p){
    while(p>9) p/=10;
    return p;
}
int main(){
    printf("p1i--Make a multiplication table"
        " and count leading digits.\n\n");
    for(int i=1;i<10;i++){
        for(int j=1;j<10;j++){
            int p=i*j;
            printf("%3d ",p);
            count[leading(p)]++;
        }
        printf("\n");
    }
    printf("\n#%10s %10s %10s\n","digit","count","percent");
    for(int i=1;i<10;i++) {
        printf(" %10d %10d %10.2f\n",i,count[i],
            100.0*count[i]/(9*9));
    }
    return 0;
}
```

When run it produced the output

```
p1i--Make a multiplication table and count leading digits.

  1   2   3   4   5   6   7   8   9
  2   4   6   8  10  12  14  16  18
  3   6   9  12  15  18  21  24  27
  4   8  12  16  20  24  28  32  36
  5  10  15  20  25  30  35  40  45
  6  12  18  24  30  36  42  48  54
```

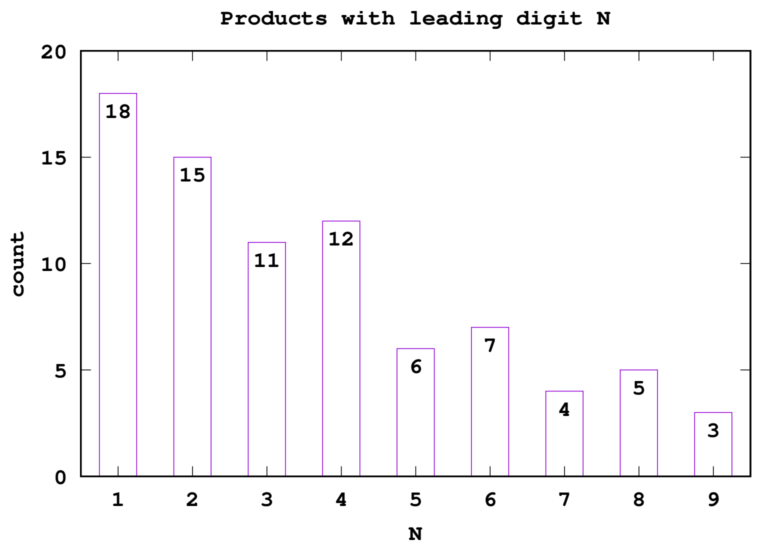
```
   7  14  21  28  35  42  49  56  63
   8  16  24  32  40  48  56  64  72
   9  18  27  36  45  54  63  72  81

#     digit      count    percent
          1         18      22.22
          2         15      18.52
          3         11      13.58
          4         12      14.81
          5          6       7.41
          6          7       8.64
          7          4       4.94
          8          5       6.17
          9          3       3.70
```

The Gnuplot script

```
set terminal postscript color enhanced eps font "Courier-Bold"
set output 'p1i.eps'
set size 0.7,0.7
set style histogram cluster gap 1
set style data histogram
set key left
set ylabel "count"
set xlabel "N"
set title "Products with leading digit N"
plot [0.5:9.5] [0:20] \
    newhistogram at 1, \
    "p1i.dat" using 2:xtic(1) lt 1 lc 1 title "",\
    "" using ($1):($2-.8):2 with labels ti ""
```

then produced the bar chart



(ii) Write a computer program compute all products of the form

$$d_1 \cdot d_2 \cdot d_3 \qquad \text{where} \qquad d_i \in \{1, 2, \ldots, 9\}$$

and count how many begin with the digit 1, how many with a 2 and so forth up to 9. Also compute the percentages in each category. For reference, the output of your program should look like

```
#     digit      count    percent
          1        218      29.90
          2        137      18.79
          3         94      12.89
          4         81      11.11
          5         46       6.31
          6         43       5.90
          7         37       5.08
          8         37       5.08
          9         36       4.94
```

Please submit your source code—in any language—for this question.

I wrote a program in C that looked like

```c
#include <stdio.h>

int count[10];
int leading(int p){
    while(p>9) p/=10;
    return p;
}
int main(){
    printf("p1ii--Count leading digits in products"
        " of the form d0*d1*d2.\n");
    for(int i=1;i<10;i++){
        for(int j=1;j<10;j++){
            for(int k=1;k<10;k++){
                int p=i*j*k;
                count[leading(p)]++;
            }
        }
    }
    printf("\n#%10s %10s %10s\n","digit","count","percent");
    for(int i=1;i<10;i++) {
        printf(" %10d %10d %10.2f\n",i,count[i],
            100.0*count[i]/(9*9*9));
    }
    return 0;
}
```

which produced the output

```
p1ii--Count leading digits in products of the form d0*d1*d2.

#     digit      count    percent
          1        218      29.90
          2        137      18.79
          3         94      12.89
          4         81      11.11
          5         46       6.31
          6         43       5.90
          7         37       5.08
          8         37       5.08
          9         36       4.94
```

(iii) In the text it is suggested that the mantissas of the numbers which appear in numerical calculations follow the reciprocal distribution given by the density

$$r(x) = \frac{1}{x \log b} \qquad \text{for} \qquad x \in [1/b, 1].$$

Set $b = 10$ and calculate

$$p_d = \int_{d/10}^{(d+1)/10} r(x)dx \qquad \text{for} \qquad d \in \{1, 2, \ldots, 9\}.$$

to find the probability a numbers starts with the digit d under this hypothesis.

Using elementary calculus compute

$$p_d = \int_{d/10}^{(d+1)/10} \frac{1}{x \log 10} dx = \frac{\log(d+1) - \log d}{\log 10}.$$

To approximate p_d for $d \in \{1, 2, \ldots, 0\}$ I wrote the program

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

double p(double d){
    return log1p(1/d)/log(10);
}
int main(){
    printf("p1iii--The probabilities p(d)"
        " a number starts with digit d.\n\n");
    printf("#%10s %10s %10s\n","digit","p(d)","100*p(d)");
    for(int i=1;i<10;i++) {
        double y=p(i);
        printf(" %10d %10.5f %10.2f\n",i,y,100.0*y);
    }
    return 0;
}
```

which produced the output

```
p1iii--The probabilities p(d) a number starts with digit d.

#     digit       p(d)   100*p(d)
          1    0.30103      30.10
          2    0.17609      17.61
          3    0.12494      12.49
          4    0.09691       9.69
          5    0.07918       7.92
```

```
6    0.06695       6.69
7    0.05799       5.80
8    0.05115       5.12
9    0.04576       4.58
```

A third column has been added to the above output which converts the probabilities p_d into percentages in anticipation of the next question.

(iv) Repeat question (ii) for products of the form

$$\pi_n = d_1 \cdots d_n \qquad \text{for} \qquad n = 4, 5, 6$$

and comment on how the percents computed for different values of n compare with the theoretical probabilities in part (iii).

I wrote a program that looked like

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

double p(double d){
    return (log(d+1)-log(d))/log(10);
}
int leading(int p){
    while(p>9) p/=10;
    return p;
}
void products(int count[10],int n){
    int d[n+1];
    for(int i=0;i<=n;i++) d[i]=1;
    while(d[n]==1){
        int p=d[0];
        for(int j=1;j<n;j++){
            p*=d[j];
        }
        count[leading(p)]++;
        int j=0;
        while(d[j]==9) {
            d[j]=1; j++;
        }
        d[j]++;
    }
    return;
}
int main(){
    printf("p1iv--Compare the percents of d1*...*dn"
        " to theoretical.\n");
    int count[10];
    int s=81;
    for(int n=2;n<7;n++){
        for(int i=0;i<10;i++) count[i]=0;
        products(count,n);
        printf("\n# n=%d\n",n);
        printf("#%10s %8s %8s %9s %12s\n","digit",
            "count","percent","100*p(d)","difference");
        for(int i=1;i<10;i++) {
            double xper=100.0*count[i]/s;
            double xthy=100.0*p(i);
            printf(" %10d %8d %8.2f %9.2f %12.4f\n",i,
                count[i],xper,xthy,xper-xthy);
        }
        s*=9;
    }
    return 0;
}
```

which produced the output

```
p1iv--Compare the percents of d1*...*dn to theoretical.

# n=2
#     digit    count  percent  100*p(d)   difference
          1       18    22.22     30.10      -7.8808
          2       15    18.52     17.61       0.9094
          3       11    13.58     12.49       1.0864
          4       12    14.81      9.69       5.1238
          5        6     7.41      7.92      -0.5107
          6        7     8.64      6.69       1.9473
          7        4     4.94      5.80      -0.8609
          8        5     6.17      5.12       1.0576
          9        3     3.70      4.58      -0.8720

# n=3
#     digit    count  percent  100*p(d)   difference
          1      218    29.90     30.10      -0.1990
          2      137    18.79     17.61       1.1837
          3       94    12.89     12.49       0.4005
          4       81    11.11      9.69       1.4201
          5       46     6.31      7.92      -1.6081
          6       43     5.90      6.69      -0.7962
          7       37     5.08      5.80      -0.7237
          8       37     5.08      5.12      -0.0398
          9       36     4.94      4.58       0.3625

# n=4
#     digit    count  percent  100*p(d)   difference
          1     2035    31.02     30.10       0.9136
          2     1170    17.83     17.61       0.2235
          3      796    12.13     12.49      -0.3616
          4      629      9.59       9.69       -0.1040
          5      486      7.41       7.92       -0.5107
          6      394      6.01       6.69       -0.6895
          7      376      5.73       5.80       -0.0684
          8      371      5.65       5.12        0.5394
          9      304      4.63       4.58        0.0577

# n=5
#     digit    count  percent  100*p(d)   difference
          1    17983    30.45     30.10        0.3514
          2    10321    17.48     17.61       -0.1304
          3     7288    12.34     12.49       -0.1516
          4     5510     9.33      9.69       -0.3598
          5     4806     8.14      7.92        0.2209
          6     3920     6.64      6.69       -0.0561
          7     3451     5.84      5.80        0.0451
          8     3240     5.49      5.12        0.3717
          9     2530     4.28      4.58       -0.2912

# n=6
#     digit    count  percent  100*p(d)   difference
          1   160073    30.12     30.10        0.0176
          2    93006    17.50     17.61       -0.1084
          3    67049    12.62     12.49        0.1226
          4    50546     9.51      9.69       -0.1799
          5    43011     8.09      7.92        0.1752
          6    37583     7.07      6.69        0.3772
          7    29804     5.61      5.80       -0.1910
          8    27688     5.21      5.12        0.0947
          9    22681     4.27      4.58       -0.3079
```

Note that the difference between the theory and the percents generated from the products $d_1 \cdots d_n$ decrease as n increases.

(v) [Extra credit and for Math 666] Read the sections in our textbook about the reciprocal distribution. State which sections you read and then provide a theoretical explanation why the mantissas of the floating-point numbers which appear in typical computations follow the reciprocal distribution.

I read Section 2.8 The Frequency Distribution of Mantissas. This section proved persistence of the reciprocal distribution under multiplication, which may be stated as

> **Theorem.** *If the mantissa of either x or y is distributed according to the reciprocal distribution then the mantissa of xy is also distributed according to the reciprocal distribution.*

In that section it was also suggested and given as a homework problem that the reciprocal distribution is persistent under division. Consequently, we have

> **Theorem.** *If the mantissa of either x or y is distributed according to the reciprocal distribution then the mantissa of x/y is also distributed according to the reciprocal distribution.*

After showing these persistence results, the text also proved that even if the mantissas of neither x nor y are distributed according to the reciprocal distribution, then the mantissa of xy as well as x/y is at least no further from the reciprocal distribution than either x or y. Here, the sense of distance was defined by

> **Definition.** *Let h represents the density of the distribution of the mantissa of x. The distance of that distribution from the reciprocal distribution is defined as*
>
> $$D\{h\} = \max\left\{ \frac{h(z) - r(z)}{r(z)} : \frac{1}{b} \le z \le 1 \right\}$$
>
> *where $r(x) = 1/(x \log 10)$ is the probability density of the reciprocal distribution.*

While no theoretical estimates were given showing how fast the distribution of mantissas converge to the reciprocal distribution under products or quotients, some calculation results were provided when the distribution of x and y originally followed a uniform distribution. Those results were similar to the computations performed in the earlier parts of this question. Note that no persistence results related to addition or subtraction were shown. Therefore, it appears the reciprocal distribution only arises from the process of taking repeated products and quotients.

2. This question explores how rounding errors accumulate in a sum of numbers. To avoid the difficulties of floating-point arithmetic we consider the simpler case of fixed-point numbers of the form `X.XXXXXXX` where each `X` corresponds to a decimal digit.

(i) Suppose real numbers X_i are chosen randomly according to a uniform distribution in the interval $[0, 1]$. Let X_i^* be the result of rounding X_i to the nearest approximation of the form `X.XXXXXXX`. For definiteness, round so the last digit is even in the case of a tie. Explain why it is reasonable to assume that the resulting rounding errors

$$\varepsilon_i = X_i^* - X_i$$

will be uniformly distributed on the interval $[-0.00000005, 0.00000005]$.

The rounding function $X_i \to X_i^*$ may be written as

$$X_i^* = \begin{cases} 0.000\,000\,0 & \text{if } 0.000\,000\,00 \le X_i \le 0.000\,000\,05 \\ 0.000\,000\,1 & \text{if } 0.000\,000\,05 < X_i < 0.000\,000\,15 \\ 0.000\,000\,2 & \text{if } 0.000\,000\,15 \le X_i \le 0.000\,000\,25 \\ \quad\vdots & \qquad\qquad\qquad\vdots \\ 0.999\,999\,8 & \text{if } 0.999\,999\,75 \le X_i \le 0.999\,999\,85 \\ 0.999\,999\,9 & \text{if } 0.999\,999\,85 < X_i < 0.999\,999\,95 \\ 1.000\,000\,0 & \text{if } 0.999\,999\,95 \le X_i \le 1.000\,000\,00. \end{cases}$$

There are special cases at the endpoints of the interval $[0, 1]$ such that numbers can only round down to 0 and only up to 1. Otherwise, we obtain either

$$X_i \in [X_i^* - 0.000\,000\,05, X_i^* + 0.000\,000\,05]$$

or

$$X_i \in (X_i^* - 0.000\,000\,05, X_i^* + 0.000\,000\,05)$$

and since the X_i are uniformly distributed, then conditioning on each interval—whether or not the endpoints are open or closed—yields that X_i are uniformly distributed on the closed version of that same interval. Thus, $X_i^* - X_i$ has been seen to be uniformly distributed on the interval $[-0.000\,000\,05, 0.000\,000\,05]$ except in the case when X_i^* is either 0 or 1.

To handle this final case, note that the uniform distribution of X_i also implies

$$\mathbf{P}\{0.000\,000\,00 \le X_i \le 0.000\,000\,05\} = \mathbf{P}\{0.999\,999\,95 \le X_i \le 1.000\,000\,00\}.$$

Now, after conditioning on the possibilities $X_i^* = 0$ or $X_i^* = 1$ together, the chances of rounding up or down are equally likely. In particular, we find that $X_i^* - X_i$ is again uniformly distributed on the interval $[-0.000\,000\,05, 0.000\,000\,05]$. Having treated all cases, the general result now holds.

(ii) Statistical simulations can be performed on a computer by specifying a seed and then using a pseudo-random number generator to create a sequence of numbers based on that seed. Let $\mathcal{S}$ be the set of seeds defined as

$$\mathcal{S} = \{\, p(i) : i = 1, \ldots, 20 \,\} \qquad \text{where} \qquad p(x) = x^3 + x + 1.$$

Write a program to compute and print all twenty seeds.

I wrote the program

```c
#include <stdio.h>

int p(int i){
    return (i*i+1)*i+1;
}
int main(){
    printf("p2ii--Print all twenty seeds"
        " for a statistical simulation.\n\n");
    printf("#%9s %12s\n","i","p(i)");
    for(int i=1;i<=20;i++){
        printf("%10d %12d\n",i,p(i));
    }
    return 0;
}
```

which produced the output

```
p2ii--Print all twenty seeds for a statistical simulation.

#        i         p(i)
         1            3
         2           11
         3           31
         4           69
         5          131
         6          223
         7          351
         8          521
         9          739
        10         1011
        11         1343
        12         1741
        13         2211
        14         2759
        15         3391
        16         4113
        17         4931
```

```
18          5851
19          6879
20          8021
```

There wasn't much to do so I checked a few of these by hand and they seem fine.

(iii) The following C program generates n numbers uniformly distributed on the interval $[-0.00000005, 0.00000005]$ based on the `seed` specified in line 5.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 int main(){
 4     int n=4,seed=992314;
 5     srandom(seed);
 6     printf("seed %d:",seed);
 7     for(int i=0;i<n;i++){
 8         double epsilon=0.0000001*random()/RAND_MAX-0.00000005;
 9         printf(" %g",epsilon);
10     }
11     printf("\n");
12     return 0;
13 }
```

Modify the above program or write your own to print the first four numbers corresponding to each `seed` $\in \mathcal{S}$. Note that the results may depend on what language, computer and operating systems you choose to use when answering this question. Your report should include source code as well as output.

I modified the code to obtain

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

int p(int i){
    return (i*i+1)*i+1;
}
int main(){
    printf("p2iii--Print four numbers corresponding"
        " to each seed in S.\n\n");
    for(int j=1;j<=20;j++){
        int n=4,seed=p(j);
        srandom(seed);
        printf("seed %d:",seed);
        for(int i=0;i<n;i++){
            double epsilon=0.0000001*random()/RAND_MAX-0.00000005;
            printf(" %g",epsilon);
        }
        printf("\n");
    }
    return 0;
}
```

which produced the output

```
p2iii--Print four numbers corresponding to each seed in S.

seed 3: 6.13802e-09 -2.75017e-08 -1.06908e-08 -5.60616e-09
seed 11: 4.26345e-08 2.64997e-09 -1.90219e-08 2.2105e-10
seed 31: 9.2119e-09 2.87117e-08 3.62696e-08 -9.77059e-09
seed 69: 2.10721e-08 4.83184e-08 3.61917e-08 6.65504e-09
seed 131: 4.27864e-08 -4.0747e-08 1.23879e-08 -8.96591e-09
seed 223: -1.01338e-08 -4.05078e-08 4.66581e-08 -1.39903e-08
seed 351: 2.73462e-08 4.71206e-08 -2.89584e-08 -1.63243e-08
seed 521: 1.90335e-08 -3.14898e-08 1.51742e-08 -2.53968e-08
seed 739: 3.22152e-08 -2.59194e-08 1.33286e-08 3.55071e-09
seed 1011: 2.99607e-08 9.4839e-09 4.50413e-08 1.02699e-08
seed 1343: 2.83515e-08 -2.61591e-08 -6.74133e-09 -1.19148e-08
seed 1741: -7.87187e-09 -3.54336e-08 3.89929e-08 2.87061e-08
seed 2211: -1.40778e-08 -2.10627e-08 1.31047e-08 2.37121e-08
seed 2759: 2.48398e-08 1.5139e-08 -1.95165e-09 1.53095e-08
seed 3391: -2.50259e-08 -2.77288e-08 2.6146e-08 -3.21723e-09
seed 4113: 4.954e-08 -3.87762e-09 2.69852e-08 8.04799e-09
seed 4931: -3.53068e-08 3.53229e-08 -1.67706e-08 -7.46813e-09
seed 5851: 3.63764e-08 3.88701e-08 2.75082e-08 4.33772e-08
seed 6879: -2.28735e-08 -4.77052e-08 -1.12922e-08 -3.40262e-10
seed 8021: -4.52227e-08 2.64932e-08 -4.77733e-08 -4.31564e-08
```

Note that the four pseudo-random numbers corresponding to each of the 20 different seeds are different. It is also possible, if you used a different computer or programming language that all the numbers are different than what is reported above. In this case, the numbers should all at least lie in the interval $[-0.000\,000\,05, 0.000\,000\,05]$ and be different.

(iv) The sum of n rounding errors may be simulated by computing

$$E_n(\texttt{seed}) = \sum_{i=1}^{n} \varepsilon_i$$

where ε_i is the sequence of pseudo-random numbers corresponding to `seed` from part (iii) of this question. The root-mean-squared average

$$R_n = \Big(\frac{1}{20} \sum_{\texttt{seed} \in \mathcal{S}} \big|E_n(\texttt{seed})\big|^2 \Big)^{1/2}$$

can be used to characterize the expected error after n additions. Write a program to compute R_n for $n = 2^k$ with $k = 2, \ldots, 20$. Include the source code and output.

I wrote the program

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

int p(int i){
    return (i*i+1)*i+1;
}
double En(int n,int seed){
    srandom(seed);
    double E=0;
    for(int j=0;j<n;j++){
        double epsilon=0.0000001*random()/RAND_MAX-0.00000005;
        E+=epsilon;
    }
    return E;
}
double Rn(int n){
    double R=0;
    for(int i=1;i<=20;i++){
        double E=En(n,p(i));
        R+=E*E;
    }
    return sqrt(R/20);
}
int main(){
    printf("p2iv--Print Rn for n=2^k"
        " where k=2,...,20.\n\n");
    int n=2;
    printf("#%7s %10s %14s\n","k","n","Rn");
    for(int k=2;k<=20;k++){
        n*=2;
        printf("%8d %10d %14.5e\n",k,n,Rn(n));
    }
    return 0;
}
```

which produced the output

```
p2iv--Print Rn for n=2^k where k=2,...,20.

#       k          n             Rn
        2          4    6.40086e-08
        3          8    9.55103e-08
        4         16    1.33514e-07
        5         32    1.88863e-07
        6         64    2.93637e-07
        7        128    3.19049e-07
        8        256    3.96978e-07
        9        512    3.85517e-07
       10       1024    8.54399e-07
       11       2048    1.43634e-06
       12       4096    2.07374e-06
       13       8192    2.90232e-06
       14      16384    3.61452e-06
       15      32768    4.26340e-06
       16      65536    5.85567e-06
       17     131072    1.06668e-05
       18     262144    1.36089e-05
       19     524288    1.90202e-05
       20    1048576    2.69180e-05
```

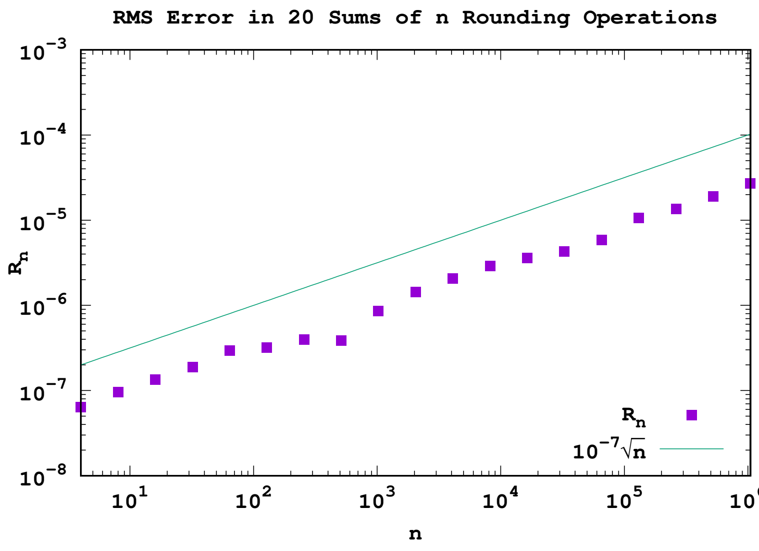
Note that the error root-mean-squared error given by R_n increases as n increases.

(v) Show the accumulation of rounding error simulated by your program increases as $\sqrt{n}$ where n is the number of terms in the sum by making a log-log plot of (n, R_n) compared with the function `1e-7*sqrt(x)`.

The Gnuplot script

```
set terminal postscript color enhanced eps font "Courier-Bold"
set output 'p2v.eps'
set size 0.7,0.7
set key bottom right
set key spacing 1.5
set ylabel "R_n"
set xlabel "n"
set format y "10^{%L}"
set format x "10^{%L}"
set logscale x
set logscale y
set title "RMS Error in 20 Sums of n Rounding Operations"
plot [] [] \
    "p2iv.dat" using 2:3 with points pt 5 title "R_n",\
    1e-7*sqrt(x) ti "10^{-7}{/Symbol @{\140}\326}n"
```

created the graph



Since the slope of the line and the green dots are roughly the same, one can conclude that the accumulation of rounding errors simulated by the program increase as $\sqrt{n}$.

(vi) [Extra credit and for Math 666] Read the Wikipedia article on the random walk and any other source of information which you find useful. Provide references to what you read and then give a theoretical explanation why the accumulation of rounding errors should grow as $\sqrt{n}$ where n is the number of terms in the sum.

The Wikipedia article was at

`https://en.wikipedia.org/wiki/Random_walk`

Since rounding error is with equal probability positive or negative, we compare the sum $E_n(\texttt{seed})$ to a one-dimensional random walk of the form

$$S_n = \sum_{i=1}^{n} Z_i$$

where Z_i are independent random variables that are either 1 or -1 with a 50 percent probability for either value. Probability shows that

$$\mathbf{E}[S_n] = \sum_{i=1}^{n} \mathbf{E}[Z_i] = 0$$

while independence of the Z_i shows

$$\mathbf{E}[S_n^2] = \mathbf{E}\Big[\sum_{i=1}^{n}\sum_{j=1}^{n} Z_iZ_j\Big] = \sum_{i=1}^{n}\mathbf{E}[Z_i^2] + \sum_{i\neq j}\mathbf{E}[Z_iZ_j] = \sum_{i=1}^{n}\mathbf{E}[Z_i^2] = n.$$

It follows that the expected root-mean-square error is

$$\sqrt{\mathbf{E}[S_n^2]} = \sqrt{n}.$$

We now adapt the above argument to consider case where Z_i are independently distributed rounding uniformly distributed on the interval $[-\delta, \delta]$ where $\delta = 0.000\,000\,05$. In this case

$$\mathbf{E}[Z_i^2] = \frac{1}{2\delta}\int_{-\delta}^{\delta} x^2 dx = \frac{1}{6\delta}x^3\Big|_{-\delta}^{\delta} = \frac{1}{3}\delta^2.$$

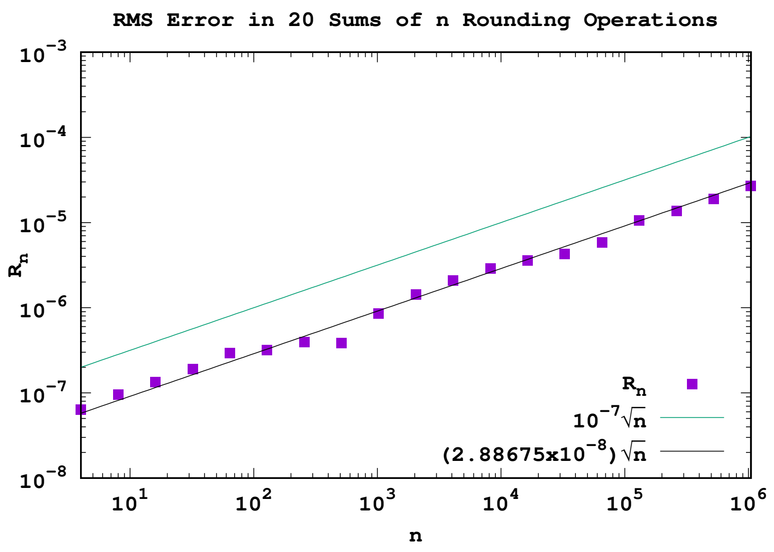
Consequently

$$\mathbf{E}[S_n^2] = \sum_{i=1}^{n}\mathbf{E}[Z_i^2] = \frac{n}{3}\delta^2$$

and

$$R_n \approx \Big(\frac{n}{3}\Big)^{1/2}(0.000\,000\,05) = K\sqrt{n} \qquad \text{where} \qquad K = 2.88675\times 10^{-8}.$$

We add the curve $K\sqrt{n}$ to the plot in the previous part of this question to check whether the theoretical value of K computed above is reasonable. The resulting graph



is a strong indication that the above analysis is correct.