Computational Science Basics and Error and Uncertainty

Mitchell Miller

Physics 240

In this assignment, several properties of computational science were explored. First, the method of numerical storage in computer memory was analyzed. Using iterative loops, both underflow and overflow limits were easily calculated. In addition to these limits, the precision at which a computer can store different types of numbers was found, again using iterative loops. In order to explore how these limits affect calculations, the Taylor Series expansion for sin(x) was calculated. Next, the concept of subtractive cancellation was explored by manipulating three different series to obtain the most accurate results possible. The direction of summing, upward or downward, and their affects on error were then examined using spherical Bessel functions. Finally, again using a Taylor Series approximation for the sin(x) function, some methods of error avoidance were tested.

1. Introduction

Since computers must store numbers in physical memory space, the length and precision of these stored items is quite limited. This imposes a number of other restrictions on numerical manipulations as well. When using both very large and very small numbers, computers lose the ability to perform accurate calculations. Additionally, error can appear as a result of several other calculations that one would not expect. In order to effectively design and execute programs to solve problems, it is vital for a physicist to understand the cause of these errors and posses knowledge of methods to avoid them. By using very simple methods, one can avoid a significant amount of error. Simple algebraic manipulation of sums or complex calculations to eliminate terms that become very large or very small can reduce a great amount of error. Furthermore, by simplifying equations to remove the subtraction of nearly equivalent terms can also reduce error. Since these conditions apply to many real world problems, ranging from complex kinematic systems to thermodynamics to quantum mechanics, the ability to recognize these errors and avoid them will be invaluable.

2. Theory

Data storage is done by writing information to bits. A bit is either a high or low voltage, which corresponds to a 1 or 0, respectively, in binary. The size of memory is most commonly described in bytes, where 1 byte is equal to 8 bits. Real numbers are stored as what is called a ‘floating-point number’, which is the binary equivalent of scientific notation or as ‘fixed-point numbers’ when the number is an integer or has a fixed number of decimal points. Most scientific calculations require floating-point numbers. Floating-point numbers are stored as a 9 digit mantissa, an exponent, and a sign bit. The number 123456, for example, could be written as 1.23456E+5. In this case, the mantissa would be ‘1.23456’ and the exponent ‘5’. The standard form for storing a floating point is

$$x_{float} = (-1)^s * 1.f * 2^{e-bias} \qquad (1)$$

This means any number X_{float} is stored as negative one raised to the sign bit, 0 for positive and one for negative, and then multiplied by 1.f. For normal numbers, the first digit is always assumed to be one, which allows the storage of only the fractional part, 'f'. This is then all multiplied by 2 raised to the exponent minus a bias. The exponent is 1 byte, which means it can range from 0 to 255, with these two endpoints being a special case. When e is 255, the output is either positive or negative infinity, depending on the sign bit. When e is 0, the output is either zero when 'f' is 0 or, when 'f' is nonzero, the one in 1.f is set to a zero. Finally, the bias, used to keep the stored bias always positive, is 127 for normal numbers, and 126 for 'subnormal' numbers which have an exponent of 0 and a nonzero fractional part. For single point floats these values are all stored in a 32 bit word with 's' being bit 31, 'e' being bits 30 to 23, and 'f' being bits 22 to 0.

One common form of error is caused by the fact that there are a finite number of values that can be stored called 'machine numbers.' Any number that is stored that does not exactly match one of these values is truncated to the closest machine number. Since there is only room for nine digits in the mantissa, anything longer will cause overflow or underflow, meaning a number such as 1111111111111 will be stored as 1111111110000. It is quite easy to see how this could cause error in even the simplest calculations.

Another common cause of error is machine precision. When two numbers with different exponents are added, the exponent of the smaller number is made large while the mantissa is made smaller. To do this, the mantissa is shifted right by inserting zeroes until the numbers being added have matching exponents. If this exponential difference is very large, the smaller number can be entirely or partially lost due to the bit shifting.

Approximation error is caused by the simplification of mathematic functions for use on computers. A simple example of this is the sin(x) function

$$\sin x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} \qquad (2)$$

In order to calculate this with a calculator or a computer, one must use a series approximation. A series will give the exact value for the function given an infinite number of terms, but this is obviously impossible. This type of error can be reduced by increasing the number of terms to a large value.

Subtractive error can also cause significant unexpected error. When two large values that are close together are subtracted, the result can lose a large amount of accuracy. The example explored in this experiment is the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad (3)$$

$$x = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}} \qquad (4)$$

In addition to these two equations, two other series were examined

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1} \qquad (5)$$

$$S^{(up/down)} = \sum_{n=1}^{N} \frac{1}{n} \qquad (6)$$

A Bessel Function is a regular solution to the differential equation

$$x^2 \frac{d^2 y}{dx^2} + 2x \frac{dy}{dx} + [x^2 - l(l+1)]f(x) = 0 \qquad (7)$$

The resulting equation can provide a value for any given value of x, provided the following via upward or downward recursion

$$J_0 = \frac{\sin x}{x}, \qquad J_1 = \frac{\sin x}{x^2} - \frac{\cos x}{x} \qquad (8)$$

$$J_l(x) = \frac{2l-1}{x} J_{l-1}(x) - J_{l-2} \; Upward \qquad (9)$$

$$J_l(x) = \frac{2l+3}{x} J_{l+1}(x) - J_{l+2} \; Downward \qquad (10)$$

By comparing the two methods, one sees the result of subtractive cancelling clearly. By subtracting the two previous *J* terms in the upward method to get a smaller term, one is left with highly erroneous result. When using the downward method, two small values are combined to get larger values.

3. Experimental Method

The first part of this lab examines underflow and overflow of floats, doubles, and integers. This is done by doubling and halving two different values repeatedly. After a certain number of loops, the computer can no longer precisely store the values and assumes them to be infinity or zero. For integers, one was added and subtracted from two values.

The next section seeks to determine machine precision. To do this, a value is added to one. The value that is added to one is then halved and added again to one. When the value is halved enough times and added to one, the output eventually becomes just one. This is the point where the halved value is so small that it can no longer be stored.

The next part of this lab examines the effects of subtractive cancellation. First, the results of the quadratic formula, equations (3) and (4), and the resulting error were observed. Next, the sum of equation (5) was examined. By manipulating this equation, one can eliminate the subtraction

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)} \qquad (13)$$

This prevents the subtractive cancellation, increasing accuracy. Finally, this part examines the sums in equation (6). By comparing the results of this sum for upward and downward methods, relative error can be determined. The downward summation generates more accurate results due to the fact that it adds small terms to get a large one.

4. Data and Analysis

For the first part of this lab, underflow and overflow limits were calculated. For floats, overflow occurs at just below 3.4E+38 and underflow at 1.4E-45. For a double, the overflow limit is 1.8E+308 and underflow at 4.9E-324. Integers do not overflow or underflow in the same manner as the other variables. When the high value reaches 2147483647 and is increased, it rolls over to -214748648. The same occurs for negative values when they reach the limit, rolling over to the positive value.

Next, the machine precision was found. For float, the minimum value that can be added effectively is slightly less than 1.1E-7. For doubles, the value is just below 2.0E-16.

The next section explores the effects of subtractive cancellation. Using equations (3) and (4), the effects were observed to happen when c ≈ 1E-12 when the plus/minus is subtraction. The next part involves the manipulation of equation (5) into (13). By begin with N = 1 and increasing up to N = 1,000,000, the following log-log plot of error vs counts is produced.

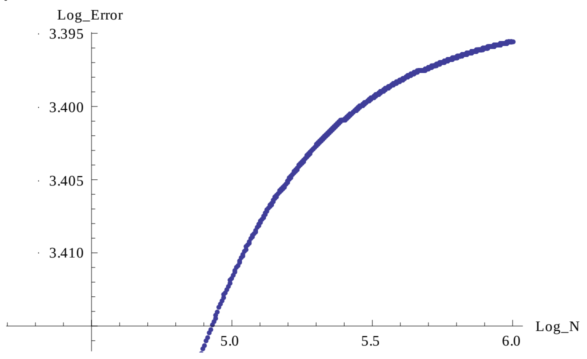

Figure 1: Log-Log plot of Error VS number of iterations.

This is a clear curve toward a limit value.

Finally, equation (6) was calculated using both upward and downward methods. The upward and downward methods were compared to eachother and the relative error found. This was plotted on a log-log scale as shown if Figure 2.

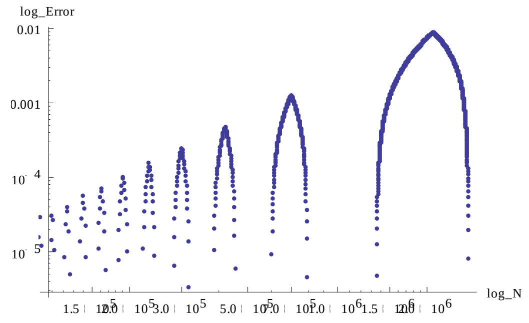

Figure 2: Log-Log plot of Relative error VS number of iterations.

This plot has a strange cyclic nature, showing repeatedly increasing and decreasing error, with the overall trend being upward.

5. Conclusion

This lab successfully demonstrated the causes and effects of various forms of errors and uncertainties. First, overflow and underflow errors were examined. Next, the affects of approximation error was explored. Finally, subtractive cancellation was observed and methods for avoiding these errors were found.

6. Bibliography

[1] "Bessel Function of the First Kind -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 29 Jan. 2010. <http://mathworld.wolfram.com/BesselFunctionoftheFirstKind.html>.
[2]"C - fwrite, fscanf and fprintf." Web. 29 Jan. 2010. <http://irc.essex.ac.uk/www.iota-six.co.uk/c/i4_fwrite_fscanf_fprintf.asp>.
[3]"The GNU C Library - Low-Level Arithmetic Functions." *U of U School of Computing*. Web. 29 Jan. 2010. <http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_18.html>.
[4]"Taylor Series -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 29 Jan. 2010. <http://mathworld.wolfram.com/TaylorSeries.html>.
[5]"Tutorial to Understand IEEE Floating-Point Errors." *Microsoft Support*. Web. 29 Jan. 2010. <http://support.microsoft.com/kb/42980>.

Appendix A: Source Code

```
#include <stdio.h>

//Exercise 1.5.2 #2
//Mitchell Miller
int main() {
	//Setting variables
	float underFloat=1;
	float overFloat=1;
	double underDouble=2.072262e-317;
	double overDouble=5.357543e+300;
	int underInt=-2147483500;
	int overInt=2147483500;
	int loopCount=1;
	int numberOfLoops=200;
	int numberOfLoopsDouble=30;
	int numberOfLoopsInt=160;

	//Test for underflow and overflow limits on float
	printf("Testing underflow and overflow limits for a floating point number \n");
	printf("Count, Under, Over \n");
	for(loopCount=1; loopCount<numberOfLoops; loopCount++){
		overFloat = overFloat*2;
		underFloat = underFloat/2;
		printf("%d %e %e \n", loopCount, underFloat, overFloat);
	}

	//Test for under and overflow limits on double
	printf("Testing underflow and overflow limits for a double \n");
	printf("Count, Under, Over \n");
	//Reset values
	loopCount=1;
	for(loopCount=1; loopCount<numberOfLoopsDouble; loopCount++){
		overDouble = overDouble*2;
		underDouble = underDouble/2;
		printf("%d %e %e \n", loopCount, underDouble, overDouble);
	}

	//Test for under and overflow limits on integer
	printf("Testing underflow and overflow limits for a floating point number \n");
	printf("Count, Smallest, Largest \n");
	//Reset count
	loopCount=1;
	for(loopCount=1; loopCount<numberOfLoopsInt; loopCount++){
		overInt = overInt + 1;
		underInt = underInt - 1;
		printf("%d %i %i \n", loopCount, underInt, overInt);
	}
	getchar();
	return 0;
}


#include <stdio.h>

//Exercise 1.5.4
//Mitchell Miller
int main() {
	//Initialize variables
	float epsFloat=1;
	float oneFloat=1;
	double epsDouble=1;
	double oneDouble=1;
	int loopCount=1;
	int numberOfLoops=100;

	//Begin precision test for floating point
	for(loopCount=1; loopCount<numberOfLoops; loopCount++){
		epsFloat = epsFloat / 2;
		oneFloat = 1. + epsFloat;
		printf("%d %.15f \n", loopCount, oneFloat);
	}

	//Begin precision test for double
	for(loopCount=1; loopCount<numberOfLoops; loopCount++){
		epsDouble = epsDouble / 2;
		oneDouble = 1. + epsDouble;
		printf("%d %.20f \n", loopCount, oneDouble);
	}
	getchar();
	return 0;
}

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

//Exerciese 1.6
//Mitchell Miller
int main() {
	//Intitialize variables
	double value=0;
	double term=1;
	double sum=1;
	double eps=1E-8;					//Set to either 1E-8 or 1E-15
	int loopCount=1;
	int numberOfLoops=240;
	int n=2;
	double x=0;
	double relativeError=0;

	//Begin summing
	printf("x,\t imax,\t sum,\t |sum-sin(x)|/sin(x) \n");
	for(loopCount=1; loopCount<numberOfLoops; loopCount++){
	term = x;
	sum = x;
	n=2;
		do{
			term = (-term*x*x)/((2*n-1)*(2*n-2));					//Calculate
next term
			sum = sum + term;
//Add to total
			n = n + 1;
		} while(abs(term/sum)>eps);
	relativeError = abs(sum - sin(x)) / sin(x);					//Calculate relative
error
	printf("%.5g,\t %.3i,\t %.12g,\t %.9e \n", x, n, sum, relativeError);		//Display results
	x = x + 0.261799167;		//0.261799167 to increase by 1/12 pi, 3.141592654 to increase by pi
	}
	printf("Press any key to exit");
   getchar();
	return 0;
}

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

//Exerciese 1.6 (factorial method)
//Mitchell Miller

//Define factorial() as a function
int factorial(int N){
	double total=1.;
	int count=1;
	for(count=N; count>0; --count){
      total = total * count;
   }
   return total;
}

int main() {
	//Intitialize variables
	double value=0;
	double term=1;
	double sum=1;
```

```
        double eps=1E-8;
        int loopCount=1;
        int numberOfLoops=240;
        int n=2;
        double x=0;
        double relativeError=0;

        //Begin summing
        printf("x,\t imax,\t sum,\t |sum-sin(x)|/sin(x) \n");
        for(loopCount=1; loopCount<numberOfLoops; loopCount++){
        term = x;
        sum = x;
        n=2;
                do{
                        term = pow(-1,n-1)*pow(x,2*n-1)/factorial(2*n-1);
//Calculate next term
                        sum = sum + term;                                               //Add
to total
                        n = n + 1;
                } while(abs(term/sum)>eps);
        relativeError = abs(sum - sin(x)) / sin(x);                             //Calculate relative
error
        printf("%.5g,\t %.3i,\t %.6g,\t %e \n", x, n, sum, relativeError);      //Display results
        x = x + 0.261799167;    //change to 0.261799167 to increase by 1/12 pi, 3.141592654 to
increase by pi
        }
        printf("Press Enter to Exit");
  getchar();
        return 0;
}

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

//Exerciese 2.1.2 #1
//Mitchell Miller
int main() {
  //Initialize variables
  double a=0;
  double b=0;
  double c=0;
  double positiveX1=0;
  double negativeX1=0;
  double positiveX2=0;
  double negativeX2=0;

  //Get values from user
  printf("a = ");
```

```
    scanf("%lf",&a);
    printf("b = ");
    scanf("%lf",&b);
    printf("c = ");
    scanf("%lf",&c);

    //Calculate quadratics
    positiveX1 = (-b + sqrt((b * b) - (4*a*c))) / (2*a);
    negativeX1 = (-b - sqrt((b * b) - (4*a*c))) / (2*a);
    positiveX2 = (-2*c) / (b + sqrt((b * b) - (4*a*c)));
    negativeX2 = (-2*c) / (b - sqrt((b * b) - (4*a*c)));

    //Display results
    printf("X_1, X_2, X'_1, X'_2 \n");
    printf("%e, %e, %e, %e \n", positiveX1, negativeX1, positiveX2, negativeX2);

    printf("Press enter to exit \n");
    getchar();
    getchar();
    return 0;
}

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

//Exerciese 2.1.2 #2
//Mitchell Miller
int main() {
    //Initialize variables
    float s1=0;
    float s2Negative=0;
    float s2Positive=0;
    float s2=0;
    float s3=0;
    int loopCount=1;
    float n=1;
    float test=0;
    float error=0;
    float logError=0;
    float logN=0;
    int totalLoopCount=1;
    int totalNumberOfLoops=1000000;
    FILE *file;

    file = fopen("2122.txt", "w");                                    //Opens txt file to output data for plotting

    for(totalLoopCount=1; totalLoopCount<=totalNumberOfLoops; totalLoopCount+=1000){//Increase
number of terms in sum by 1
```

```
        //Reset variables to begin new loop
        s1 = 0;
        s2 = 0;
        s2Positive = 0;
        s2Negative = 0;
        s3 = 0;

        //Calculate S(1)
        for(n=1; n<=(2*totalLoopCount); n++){
            s1 += pow(-1, n)*(n/(n+1)); ;
        }

        //Calculate S(2)
        n=1;
        loopCount = 1;
        for(n=1; n<=totalLoopCount; n++){
            s2Negative = (2*n-1)/(2*n) + s2Negative;
            s2Positive = (2*n)/(2*n+1) + s2Positive;
        }
        s2 = s2Positive - s2Negative;

        //Calculate S(3)
        n=1;
        loopCount = 1;
        for(n=1; n<=totalLoopCount; n++){

            s3 += 1/((2*n)*(2*n+1));
            //n = n+1;
        }

        //Calculate and display output
        error = fabs(s1-s3)/s3;
        logError = log10f(error);
        logN = log10f(totalLoopCount);
        printf("%f, %f, %f, %d \n", s1, s2, s3, totalLoopCount);
        fprintf(file, "{%lf,%lf}, ", logN, logError);                    //Writes data to txt file
    }
    fclose(file);                                               //Closes file
    printf("Press Enter to Exit");

    getchar();
}

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

//Exerciese 2.1.2 #3
//Mitchell Miller
```

```
int main() {
   //Initialize variables
   float loopCount=1;
   float sUp=0;
   float sDown=0;
   float numberOfLoops=1;
   float N=1;
   int totalLoopCount=0;
   int totalNumberOfLoops=10000000;
   float Error=1;
   FILE *file;

   file = fopen("2123.txt", "w");                                    //Open file for data plotting

   for(numberOfLoops=1; numberOfLoops<=totalNumberOfLoops; numberOfLoops+=1000){
      sUp=0;
      sDown=0;

      //Begin summing
      for(N=1; N<=(numberOfLoops); N++){                                //Upward counting sum
          sUp = sUp + (1 / N);
      }
      for(N=numberOfLoops; N>0; N--){                                  //Downward counting sum
         sDown = sDown + (1 / N);
      }
      Error = ((sUp-sDown) / (sUp+sDown));
      printf("%.15e, %.15lf \n", Error, numberOfLoops);
      fprintf(file, "{%lf,%e},", numberOfLoops, Error);                    //Write data to txt file
   }
   fclose(file);                                                  //Close file
   printf("Press Enter to exit");
   getchar();
}

#include <math.h>

//Exerciese 2.2.2
//Mitchell Miller


int main() {
   //Intialize variables
   double JUp[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   double JDown[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
   int L=0;
   double x=0.1;
   int numberOfLoops=25;
   int XCount=0;
```

```
    int XLoops=3;
    int N=2;
    double n=2;
    double scale=0;
    double error=0;


    //Change values of x
    for(XCount=0; XCount<XLoops; XCount++){
      JUp[0] = sin(x)/x;
      JUp[1] = sin(x)/pow(x,2) - cos(x)/x;
      printf("J_0(%lf) Upward = %.10lf \n", x, JUp[0]);
      printf("J_1(%lf) Upward = %.10lf \n", x, JUp[1]);
      n = 2;
      //Fill array for values of J_0 to J_24 upwardly
      for(N=2; N<numberOfLoops; N++){
        JUp[N] = ((2*n-1)/x) * JUp[N-1] - JUp[N-2];
        printf("J_%i(%lf) Upward = %e \n", N, x, JUp[N]);
        n = n + 1;
      }
      //Set variables for downward recursion
      n = 22;
      N = 2;
      JDown[24] = 1;
      JDown[23] = 1;
      for(N=22; N>=0; N--){
        JDown[N] = ((2*n+3)/x) * JDown[N+1] - JDown[N+2];
        n = n - 1;
      }
      //Scale values to known set
      scale = (sin(x)/x)/JDown[0];
      for(N=0; N<numberOfLoops; N++){
        JDown[N] = JDown[N] * scale;
        printf("J_%i(%lf) Downward = %e \n", N, x, JDown[N]);
      }
      x = x*10;
    }
    //Calculate relative error
    printf("Select value for X = ");
    scanf("%lf", &x);
    for(N=0; N<numberOfLoops; N++){
      error = fabs(JUp[N]-JDown[N]) / (fabs(JUp[N]) + fabs(JDown[N]));
      printf("L = %i \t J_%i(%.1lf)Up=%lf\t J_%i(%.1lf)Down=%lf\t Error = %e \n", N, N, x, JUp[N], N, x,
JDown[N], error);
    }
    printf("Press Enter to Exit");
    getchar();
    getchar();
```

```
}

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

//Exerciese 2.3.1
//Mitchell Miller

//Define factorial() as a function
int factorial(int N){
            double total=1.;
            int count=1;
            for(count=N; count>0; --count){
      total = total * count;
   }
   return total;
}

int main() {
            //Intitialize variables
            const double pi=3.141592654;
            double value=0;
            double term=1;
            double sum=1;
            double eps=1E-8;
            int loopCount=1;
            int numberOfLoops=12;
            int n=2;
            double x=0;
            double relativeError=0;
            int m=0;
            int numberOfPi=5;

            //Begin summing
            printf("x,\t imax,\t sum,\t |sum-sin(x)|/sin(x) \n");
            for(m=0; m<numberOfPi; m++){
   if(m==0){
      x=0;
   }
   else{
       x=0.261799167;
   }
            for(loopCount=1; loopCount<=numberOfLoops; loopCount++){
            term = x;
            sum = x;
            n=2;
                        do{
                              term = pow(-1,n-1)*pow(x,2*n-1)/factorial(2*n-1);
//Calculate next term
                              sum = sum + term;                                                    //Add
to total
                              n = n + 1;
                        } while(abs(term/sum)>eps);
                        sum = sum * pow(-1,m);
            relativeError = abs(sum - sin(x+m*pi)) / sin(x+m*pi);                           //Calculate
relative error
            printf("%.5g,\t %.3i,\t %.6g,\t %e \n", x+m*pi, n, sum, relativeError);           //Display results
            x = x + 0.261799167;
//change to 0.261799167 to increase by 1/12 pi, 3.141592654 to increase by pi
            }
   }
            printf("Press any key to exit");
   getchar();
            return 0;
}
```