ROUNDING ERRORS LAB

Imagine you are traveling in Italy, and you are trying to convert 27 US Dollars into Euros. You go to the bank teller, who gives you €20.19. Your friend is with you, and she is converting $2700. You predict that she will get 100*€20.19=€2019.00, but to your dismay, she gets €2019.11. Where did those €0.11 come from?[1]

1. OBJECTIVE

In this lab, we will explore a very common error in programming: rounding errors. We will discuss how different representations of numbers in your computer can change the values that you get when doing calculations, and how these differences can have serious consequences in your applications.

2. INTRODUCTION

Intro to floating points

In most programming languages, including Java, C++, and C, there are multiple data types that are used to represent numbers. The most common are ints, doubles, and floats. These types can be signed or unsigned. We will briefly introduce these options below:

Ints - ints are used to represent integer values without any decimals
Decimal numbers can be represented in multiple ways:

- fixed-point or floating-point
- and single or double precision

Unsigned - only positive numbers can be represented
Signed - both positive and negative numbers can be represented

What is fixed point and floating point?
Fixed vs. floating point refer to the position of the decimal point in the stored value. Fixed-point means the decimal point is always in the same position, i.e. we always have the same number of digits (in binary) before and after. In base 10, this might be always storing the tens, ones, tenths, and hundredths places, so that whether you are storing 10.00 or 00.01, there are always these 4 places.

[1] A conversion rate of 0.74782 euros to a dollar was used. You calculate $27*0.74782 and realize it comes to €20.1911, but since there are no €0.001 in euros, your amount was rounded down to €20.19. This is one example

Floating point means the decimal point may be in any position of the number. The computer stores this in binary like we write scientific notation in base 10: there is a significand (the number part of scientific notation), and exponent (the power of 10 part of scientific notation), as well as sign bits for each (which we will discuss later). Consider the base 10 example 1.09×10^{3} and 3.45×10^{-5}. In decimal, the decimal point floats to different places in the number (1090 and 0.0000345).

Why are floating points useful?
Floating points allow you to represent a huge range of values that ints and fixed point doubles do not. Imagine you have 4 digits in which to represent all numbers. With integers, you can represent numbers from 0001 to 9999. With fixed point, you might get the range 00.01 to 99.99. But say you use the first 3 digits for the significand, and the 4th for the exponent. Then you can represent 0.01×10^{0} to 9.99×10^{9}, which is a much wider range!

Computer Representation
In computer memory, everything is represented in binary (base 2) instead of decimal (base 10). Each digit is either 0 or 1, and each place is a power of 2 rather than 10. So, for example, the floating point representation of 2.375×10^{0} is actually 1.0011×2^{1} in binary.

Aside: Converting to binary
Although when you declare an int or a double in a programming language, the language stores it in binary for you in memory, it is useful to understand how numbers are written in binary to avoid the types of errors that we will discuss later on.

Here is a quick tutorial of how to convert an int to binary. Just follow these steps:
1) Take the number and divide by 2
2) If the remainder is 1, the least significant bit (2^{0}) is 1. If it divides evenly, the bit is 0.
3) Repeat for the result of the division. Each bit you add will be for the next power of 2 (build the binary number from left to right). Stop when the result is 0.

Take the example 13:
  13/2 = 6 remainder 1, so our 2^{0} bit is 1.
  6/2 = 3, so our 2^{1} is 0. We now have 01.
  3/2 = 1 remainder 1, so our 2^{2} is 1. We now have 101.
  1/2 = 0 remainder 1, so our 2^{3} is 1. We now have 1101.
Since our result is now 0, we stop and our final number is 1101.
We can convert back to base 10 by multiplying the bit value in each position by 2 to the power of the position number.
  In this example, 1101 is $2^{0}*1 + 2^{1}*0 + 2^{2}*1 + 2^{3}*1 = 1+4+8 = 13$. Success!

Every integer value can be represented as an exact binary number (in the computer, every integer value in the range stated above). Consider why this is true - if we divide by 2 each

time, eventually we will get down to 0 with remainder 1 (don't believe me? Try it out a few numbers!). This will always be our most significant bit.
Note: To get the binary representation of a negative integer, we want to use the "two's complement". This means flipping all of the bits and adding 1 to the result. To show that it is negative, we add a sign bit of 1 before the binary. For example, if we want -9, we take 9=1001 in binary, flip the bits to get 0110, add 1 to get 0111, and add a 1 as the sign bit for a final 10111. To be able to understand this binary, we have to know if the binary is signed so that we look for a sign bit. Then we can do the reverse of what we just did instead of reading this as the unsigned 23.

Decimal values are trickier, since many decimals cannot be represented exactly in binary. This is not just irrational numbers like 1/7 and pi.

For the integer part of the decimal value, our conversion is the same as above. For the decimal part, however, we go through a similar process, but multiplying by 2 and building the number left to right.

1) Take the decimal and multiply by 2.
2) If you get a number <1, that bit is 0, and repeat with the resulting decimal, adding each subsequent bit to the right (starting with 2^{-1}, then 2^{-2}, 2^{-3}, etc.).
3) If you get a number >1, that bit is a 1, and repeat with the result - 1 (just the decimal of the result), adding each subsequent bit to the right.
4) If you get exactly 1, that bit is 1 and you have the exact representation.

You might be able to see why this can be an issue - while division always gets down to the base case of 0 remainder 1, with multiplication there is no obvious stopping point unless you reach 1.

Consider trying to convert 0.2 to binary. 0.2*2=0.4, so our first bit is .0. 0.4*2=0.8, so we have .00. 0.8*2=1.6, so we our next bit is 1, giving us .001. We now take 1.6-1=0.6, and do 0.6*2=1.2. This gives us .0011, and we take 1.2-1=0.2 as our next number. We are now back with 0.2, and if we continue this process we will repeat this cycle infinitely, never reaching 1 exactly. Our binary number is 0.0011001100110011..., but it has no exact representation.

This will be important to remember as we discuss floating point numbers and rounding errors in this lab!

This is where single vs. double precision floats come into play. Single precision means that the number uses one “word” of memory, which in most operating systems in 32 bits, to represent the number. A double precision uses two words, i.e. 64 bits, and therefore can be more precise.

When these values are signed, they have to split their available range between positive and negative numbers, which means that they can not reach as high a positive number as an unsigned int. This is because one of these 32 (or 64 for doubles) bits is reserved to mark whether the number is positive (0) or negative (1).

Ranges of Number Types

Type	**Min**	**Max**
Unsigned int	0	4,294,967,295
Signed int	-2,147,483,648	2,147,483,647
Signed double	-1.79769e+308	1.79769e+308

Source: defined as constants in Java with Integer.MAX_VALUE, Integer.MIN_VALUE

For a 32 bit unsigned float, there are 23 bits for the significand (in the format X.XXXXX...), a sign bit for the exponent that is 0 for positive and 1 for negative, and 8 digits for the exponent value.

If it is a signed float, there are 2 sign bits (one for the significand, one for the exponent) and one less bit used for the significand. We will continue to use the base 10 analogy in this example, since it is more familiar for many students.

For more information, check out this series of YouTube videos about floating points and their representations:
Floating Point Background: http://www.youtube.com/watch?v=svFJXukm2uE
Floating Point Example: http://www.youtube.com/watch?v=t-8fMtUNX1A

So what's the problem?

Note: for now, we assume that we have 4 decimal digits to represent a floating point number (in a computer system, they are base 2 bits instead of base 10 digits).

Imprecision: Say you wanted to represent 93,425. We can write 9.34×10^4 (93,400) or 93.5×10^4 (93,500), but not 93,425 exactly. What if we tried to do $9.34 \times 10^4 + 2.50 \times 10^1$ to get 93,425. When we add these numbers together, however, we still can only represent 3 significant digits, and we again get 9.34×10^4. This can cause significant errors in calculations. If you tried to add 2.50×10^1 to 9.34×10^4 40 times, you should get 9.44×10^4, but instead you would just get 9.34×10^4 using this logic.

Rounding Errors: Say you are trying to represent the number 1/7 in decimal. This number goes on infinitely repeating the sequence 0.142857. How would you represent this in the above representation? 1.43×10^{-1} is as close as you can get. Even with 10, 20, or 100 digits, you would have to have some rounding to represent an infinite number in a finite space. If you have a lot of digits, your rounding error might seem insignificant. But consider what happens if you add up these rounded numbers repeatedly for a long period of time. If you round 1/7 to 1.42857×10^{-1}, and add up this representation of 1/7 700 times, you should get 100. However, you instead get 9.99999×10^1.

Base 2 confusion: In binary, rounding errors occur in fractions that cannot be represented as powers of 2 (rather than base 10 like the examples above). In fact, there is no way to represent 1/10 exactly in binary, even as a floating point with 23 digits. We can get a close approximation, but this is often not good enough - as you will see in the later example of the Patriot Missile disaster, these rounding errors can sometimes have huge consequences. A power of 2 like ⅛ (i.e. 2^{-3}), on the other hand, is exactly 0.001 in binary, and there is no rounding needed.

Why should we care?

Relatively minor imprecisions and rounding errors like the ones described above can have huge impacts. In this lab, you will look at a few real world examples of how these rounding errors can make cause problems for the users of computer software. Knowing how these rounding errors can occur and being conscious of them will help you be a better and more precise programmer.

3. Exercises

3.1. Try it yourself!

Now that we've discussed the difficulty of representing 1/7 in base 10, try representing 1/7 in base 2, and add up this representation 700 times.

```
public class Fractions {

      public static void main(String[] args) {
            System.out.println("Add 1/7 700 times: " +
                                    addFraction((float)1/7, 700)); //[2]
            System.out.println("Add 1/6 600 times: " +
                                    addFraction((float)1/6, 600));
            System.out.println("Add 1/8 800 times: " +
                                    addFraction((float)1/8, 800));
      }

      private static float addFraction(float fractionToAdd,
                                           int timesAdded) {
            float total = 0;
            for (int i = 0; i < timesAdded; i++){
                  total += fractionToAdd;
            }
            return total;
      }
}
```

[2] The "(float)" in front of "1/7" is called type casting. This is necessary because in our program we can imagine 1/7 not as a number, but an operation that needs to be calculated (one divided by seven). We need to tell our program that 1 / 7 is meant to be a float number, and we do this by putting (float) in front of the 1 or in front of the 7. Otherwise, the program will think 1 over 7 is meant to be an integer, because 1 and 7 are integers, and assign the result, 0, to *fraction* (as there are 0 sevens in 1). To see this, remove the type casting and see what happens.

1) Run this program. What is the total for 1/7 added 700 times? What about 1/6 added 600 times?

2) Which one of these fractions was rounded up? Which one was rounded down?

3) What about 1/8 added 800 times. What is the answer? Why? (Hint: Think of the binary representation for powers of 2.)[3]

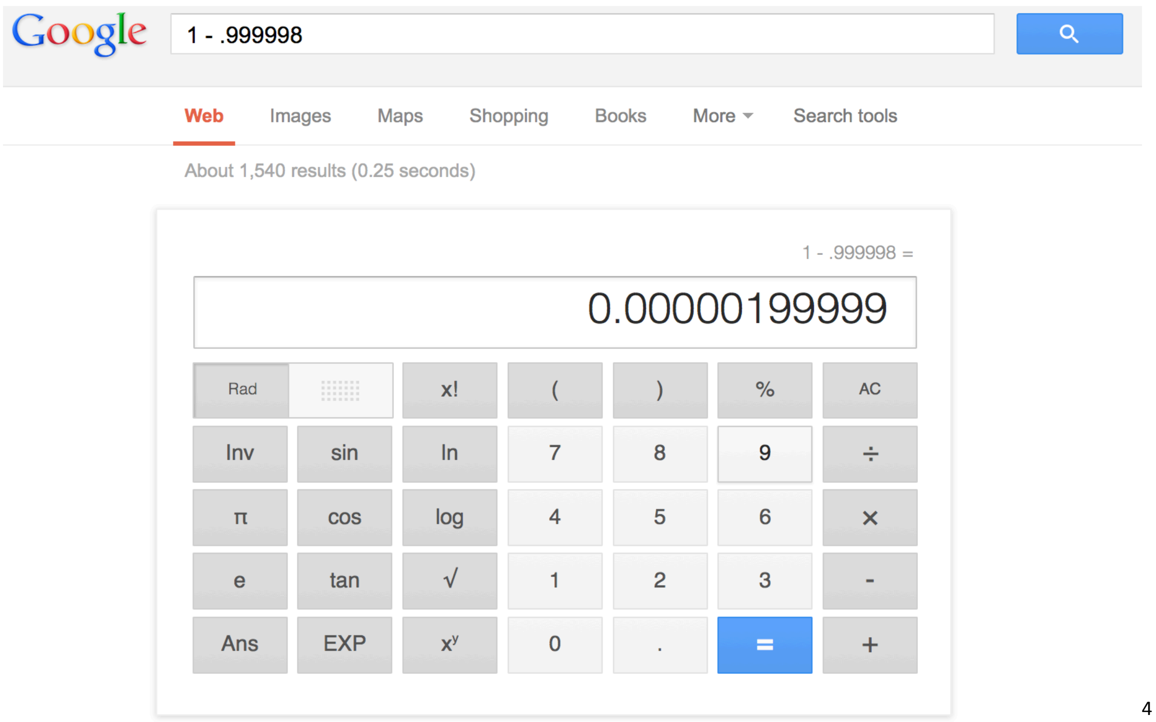
[3] A good source to convert between base 10 to base 2 is http://www.binaryconvert.com/

3.2. Google Calculator

You are in the library and need to do some calculations for your physics homework. However, you just realized that you forgot your calculator at home and decide to use Google calculator for your answers.

Here is the calculation you need to make. You have two objects. Object one traveled 1.0 meter and object two traveled .999998 meters. You want to find the difference between the two, so you go to google.com and search for: *1.0 - 0.999998*. To your dismay, Google calculator tells you that *0.00000199999* is your answer! You know that this is wrong, since the answer is clearly 0.000002.

What could have possibly gone wrong?



[4]

[4] Source: Google and the Google logo are registered trademarks of Google Inc., used with permission.

3.2.1 Representation of numbers (Google v. Bing)

Now that you found this error, you decide to try out a few more calculations in search of an answer.

First, go to Google[5] see this error with your own eyes, search for: *1.0 - 0.999998*
What do you get?

You remember that a friend told you that Bing is better at math, so you try the same search, *1.0 - 0.999998*. What do you get now?

Bing and Google show you different answers for the same calculation, but only one of the two calculations is right. What could have Bing done differently from Google in this situation?

While the actual implementation for these numbers is unknown to the public, the most likely explanation for these errors lies in the underlying binary representation of numbers by each of these search engines.

[5] To get to the Google calculator, simply go to google.com and type the equation into the search bar. If this does not bring up the Google calculator, try adding "=" (without the quotes) after the equation.

3.2.2 Write your own calculator: The difference between int, float, and double

As previously discussed, there are multiple data types that are used when representing numbers. The three main types are float, double, and int. In this section, you'll explore which type works best in different scenarios.

It might be useful to look back at the introduction and review each of these types. In this example, each of these types is signed.

In order to better understand these data types, write a program that subtracts two numbers using the three different data types described, then compare the results.

To do this, print out float1 - float2, double1 - double2, and int1 - int2. Note that in the following code, "??" are placeholders for the numbers that will be used in the exercises below.

```
public class Calculator {
     // fill in ?? for first value in below exercises
      private static final double FIRST_NUM = ??;

     // fill in ?? for second value in below exercises
      private static final double SECOND_NUM = ??;

      public static void main(String[] args) {
            System.out.println("float: " + numFloat());
            System.out.println("double: " + numDouble());
            System.out.println("int: " + numInt());
      }

      private static float numFloat(){
            float num1 = (float)FIRST_NUM;
            float num2 = (float)SECOND_NUM;
            return num1-num2;
      }

      private static double numDouble(){
            double num1 = FIRST_NUM;
            double num2 = SECOND_NUM;
            return num1-num2;
      }

      private static int numInt(){
            int num1 = (int)FIRST_NUM;
            int num2 = (int)SECOND_NUM;
            return num1-num2;
      }
}
```

1) You want to test out your calculator's limits. Try each of these functions subtracting the number 0.999999999998 from 1.0.

  a. What is the result when the numbers are represented as floats?

  b. Doubles?

  c. Ints?

2) Now, going back to your physics calculation, try to see if your calculator can get a better answer than Google did for 1.0 - 0.999998.

  a. What do you get when the numbers are represented as floats?

  b. Doubles?

  c. Ints?

3) Modify your code to perform addition instead of subtraction.

4) Try adding these numbers: 10,000,000 and 0.5

  a. What is the result when the numbers are represented as floats?

  b. Doubles?

  c. Ints?

5) Try adding these numbers: 3.25 and 0.25

  a. What is the result when the numbers are represented as floats?

  b. Doubles?

  c. Ints?

6) Try these numbers: 1,222,333 and 122,233,344

  a. What is the result when the numbers are represented as floats?

  b. Doubles?

  c. Ints?

7) Now, try experimenting with more numbers. (I.e. large integers, extremely small decimals, decimals up to hundreds place, or decimals mixed with integers.)

a. Which of these numbers seemed to be best represented by floats?

b. Doubles?

c. Ints?

d. Were your answers for floats and doubles the same or different?

8) Compare your results to those of your neighbors to see if you got different answers to any of your calculations.

As you might have noticed, ints only work for integers, while floats and doubles are better fitted for decimals. However, floats are represented differently from one computer to another. So you might or might have seen your answers for floats differ to those of your neighbors.

9) In order to solve the mystery between `float`s and `double`s, it is important to look at how many bits of information each type stores. To do so, print out the size of `int`, `float`, and `double`. You can do this by using `SIZE`, which returns the size of each of these data types in bytes.

```
public class Size {
   public static void main(String[] args) {
      System.out.println("int: " + Integer.SIZE);
      System.out.println("float: " + Float.SIZE);
      System.out.println("double: " + Double.SIZE);
   }
}
```

a. How many bits does an `int` keep track of?
b. What about a `float`?
c. `double`?

If the size for floats and doubles are the same on your computer, then the answers for your previous solutions (Previous exercises 1-7) were the same for floats and doubles. On the flip side, if the size for floats and doubles are different, your answers to your previous solutions were also different. This is because some machines store 32bits for the type float, while others will store 64bits.

b. Explain how storing 32 bits versus 64 bits for the type float will cause your answers to be different when doing arithmetic with small numbers.

3.3. Patriot Missile Failure

While some of these rounding errors seem to be very insignificant, as described by the accounting and Euro conversion examples, small errors can quickly add up. One of the most tragic events caused by rounding errors was the Patriot Missile Crisis in 1991.[6]

7

Patriot Missiles, which stand for Phased Array Tracking Intercept of Target, were originally designed to be mobile defenses against enemy aircraft. These missiles were supposed to explode right before encountering an incoming object. However, in Feb. 25, 1991, a Patriot Missile failed to intercept an incoming Iraqi Scud missile, which struck an army barrack in Dhahran. This killed 28 American soldiers and injured around 100 other people!

[6] Source: "The Patriot Missile Failure", University of Minnesota. 4 Feb 2013. http://www.ima.umn.edu/~arnold/disasters/patriot.html.
[7] Source: [Photograph by officer or employee of United States Government]. 22 April 2006 [Photograph]. Patriot Missile PAC-2. Retrieved 5 March 2013. http://commons.wikimedia.org/wiki/File:Patriot_08.jpg#filelinks

What went wrong?

The system's internal clock recorded passage of time in tenths of seconds. However, as explained earlier, 1/10 has a non-terminating binary representation. This is an approximation of what 1/10 looks like in binary:

0.0001100110011001100110011001100...

The internal clock used by the computer system only saved 24 bits. So this is what was being saved every 1/10 of a second:

0.00011001100110011001100

Chopping off any digits beyond the first 24 bits, introduced an error of about:

0.00000000000000000000000011001100
which is about 0.000000095 seconds for each 1/10 seconds.

This Patriot battery had been running for about 100 hours before the incident. Imagine this error adding up 100 hours, 10 times for each second!

1) How does this small error add up in 100 hours? Calculate the total rounding error for this missile.

The small error for each 1/10 of a second was not believed to be a problem, for the Patriot Missiles were not supposed to be operated for more than 14 hours at a time. However, the Patriot battery had been running for over 100 hours, introducing an error of about 0.34s!

Given that the Patriot Missile was supposed to intercept a Scud traveling 1,624 meters per second, 0.34 seconds was a huge problem!

2) A Scud travels 1624 meters/second, how much would the Scud have advanced in the 0.34 seconds?

The Scud traveled more than half a kilometer in this amount of time. Therefore, the Scud was out of the Patriot Missiles range, hitting the army barrack in Dhahran.

3) Now that you know more about rounding errors. Is there a way this catastrophe could have been prevented? What about trying to count steps by a different fraction? What would be an ideal fraction close to 1/10 that could have solved this problem?

3.3.1 Patriot Missile. Programming example.

Oh how exciting! It looks like you have traveled back in time with your computer and you need to write the code for the Patriot Missile. You are using the same computer you have right now, and because you are doing this before taking this class, you decide to count time in 1/10 again.

```
public class PatriotMissile {
   private static final int SECONDS = 360000;
   // 100 hours, 60 min/hr, 60 sec/min = 360000

   private static final int M_PER_S = 1624; // meters per second

   public static void main(String[] args) {
      float total = 0;

      for (int i = 0; i < SECONDS * 10; i++){
         total += (float).1;
      }

      System.out.println("Total number of seconds recorded: " +
                         total);
      System.out.println("Correct number of seconds in 100 hours: "
                         + SECONDS);

      float difference = SECONDS - total;

      System.out.println("Number of seconds your calculations are off: " +
                         difference);

      float meters = difference*M_PER_S;
      System.out.println("Meters traveled by scud in time difference: " +
                         meters);

   }
}
```

1) Run this program. What is the distance traveled by the Scud in the amount of time the Patriot Missile was off?

2) You know better than this! You go back in time, again! And decide to re-write this code. What value are you going to use instead of 1/10?

3) What happens now? What is the amount traveled by the Scud in the amount of time your Patriot Missile calculated wrong?