Assignment 2

Calculating with Floating-Point Numbers

EC602 Fall 2016

Contents

1 Introduction

1.1 Assignment Goals

The assignment goals are to

- introduce the capabilities and limitations of floating-point numbers
- provide practice on calculations with units

- provide practice with estimating error bounds

1.2 Due Date

This assignment is due 2016-09-19 at midnight

1.3 Submission Link

You can submit here: week 2 submit link

2 Background: Floating Point Numbers

Please read about floating point numbers here: IEEE floating point

In this assignment, we will focus on three floating point number representations:

- `half` / `binary16`
- `single` / `binary32`
- `double` / `binary64`

Please also see IEEE 754-1985. This standard is superseded, but there are some nice explanations at this page.

2.1 Floating Point Numbers in C++

In C++, the commonly used floating point number types are `float` which is normally a `binary32` or single-precision number, and `double` which is normally a `binary64` or double-precision number.

2.2 Floating Point Numbers in Python

In python, floating point numbers have the type `float` and are normally `binary64` or double-precision numbers.

It is possible when using the `numpy` library to define other kinds of floating-point numbers, but we won't deal with that in this assignment.

3 Examples

3.1 Effect of different precision

Here is a C++ program which shows how the value 1/3 is different depending on how you store it.

```
#include <iostream>
#include <iomanip>
#include <cassert>

using namespace std;


int main()
{
    float num_f = 1.0/3;
    double num_d = 1.0/3;

    long double num_ld = 1;
    num_ld /= 3;

    long double num_ld2 = (long double)1.0 /3;

    long double num_ld3 = 1.0 /3;

   cout << setprecision(22) << num_f << endl;
   cout << setprecision(22) << num_d << endl;
   cout << setprecision(22) << num_ld << endl;


   cout << num_d - num_f << endl;

   cout << num_ld - num_d << endl;

   cout << num_ld - num_ld2 << endl;

   cout << num_ld - num_ld3 << endl;

   cout << (long double) 1.0 - ((long double) num_f) * 3 << endl;
   cout << (long double) 1.0 - (num_f * 3) << endl;

}
```

Here is a link errors_in_floating.cpp. Try it.

3.2 Rounding Errors

If floating-point numbers were perfect, the following program would not print anything:

```
#include <iostream>
using namespace std;

int main()
{
    double one_third,zeroish;

   one_third = 1.0/3;

   for (int i=1;i<100;i++)
   {
      zeroish = 1.0 - 3.0 * (i * one_third)/ i;
      if (zeroish != 0)
         cout << i << " " << zeroish << endl;
   }

   return 0;
}
```

However, it actually prints:

```
7 1.11022e-16
14 1.11022e-16
25 1.11022e-16
28 1.11022e-16
31 1.11022e-16
50 1.11022e-16
53 1.11022e-16
56 1.11022e-16
59 1.11022e-16
62 1.11022e-16
97 1.11022e-16
```

Here is a link rounding_errors.cpp. Try it.

4 Part A: Earth as supercomputer

In this part, we get practice with units and doing floating-point calculations.

Please complete this part in both python and C++. Your goal will be to get identical results using both programs.

Suppose that the Earth is actually a giant supercomputer, and each electron represents a bit of storage (see [Hitchhikers Guide to the Galaxy]) Estimate how many electrons are on the earth, and convert this number to an equivalent number of terabytes (TB).

Your program should print out three numbers:

- your estimate in TB
- a lower bound (in TB)
- an upper bound (in TB)

So, for example, a valid print out would be

```
4.5e6
1.0e6
9.0e6
```

which means this group estimated that the Earth has a memory capacity of 4.5 million terabytes, with a lower limit of 1.0 and an upper limit of 9 million terabytes.

The filenames of the program submitted must be `w2a_earth.py` and `w2a_earth.cpp`

5 Part B: ripping apart floating point numbers

Your assignment is to complete a program that converts `float` back and forth from its parts (sign, exponent, and significant).

We show you a working version for `double`, and then a partially completed version of what you will submit for `float` or single-precision numbers.

5.1 `double_parts.cpp`

The following is a program that converts `double` back and forth from its parts (sign, exponent, and significant)

```
// double_parts
#include <iostream>
#include <iomanip>
#include <cassert>

using namespace std;

typedef unsigned long int raw64; // raw64 is a pseudonym for unsigned long int

// A structure which mimics exactly the internal representation of double
```

```
// Double Parts uses  64-bits of storage

struct Double_Parts {
    raw64 fraction : 52;  // use 52 bits for this
    raw64 exponent : 11; // then 11 bits for this
    raw64 sign : 1;      // then 1 bit for this
} ;


// these represent the positions of the SIGN, EXPONENT, and FRACTION of double.

const raw64 MASK_SIGN = 1UL << 63;
const raw64 MASK_BEXP = 0x7ffUL << 52;
const raw64 MASK_FRAC = 0xfffffffffffffUL;


// print out the parts of the structure Double_Parts
void print_dp(Double_Parts dp)
{
  if (dp.sign==1)
         cout << "negative"  << endl;
  else
        cout << "positive" << endl;

 cout << hex
      << setfill('0')
      << "expo: " << dp.exponent << endl
      << "frac: " << dp.fraction << endl
      << dec;
}


// build and take_apart are inverse functions.

Double_Parts take_apart(double d)
{
 Double_Parts dp;
 raw64 x =  *reinterpret_cast<raw64*>(&d);

 dp.sign = (x bitand MASK_SIGN) >> 63;
 dp.exponent = (x bitand MASK_BEXP) >> 52;
 dp.fraction = (x bitand MASK_FRAC);

 return dp;

}
```

```
double build(Double_Parts dp)
{
       // read this from inside out:
       // this means get the address of dp, then think of it as a pointer to a double
       // then get the double and return it.
       return *reinterpret_cast<double*>(&dp);
}

double build_alt(Double_Parts dp)
{
    raw64 c=0;

    // explicitly move the double parts to their correct locations, and add.

    c = ( (raw64)dp.sign << 63) + ( (raw64)dp.exponent << 52) + dp.fraction;

    // read this from inside out:
    // this means get the address of c, then think of it as a pointer to a double
    // then get the double and return it.
    return *reinterpret_cast<double*>(&c) ;
}

int main()
{
    assert(sizeof(raw64)==8); // make sure this is actually an 8-byte object.

    double num_from_build, num_from_build_alt;

    double numbers[5]={1.0/3,2,1e100,-5e-200,6};

    // show the structure of the numbers
    for (int i=0;i<5;i++)
    {
        // take apart the numbers, then re-build to test that it works.

        Double_Parts dp= take_apart(numbers[i]);
        num_from_build = build(dp);
        num_from_build_alt = build_alt(dp);

        cout << endl;
        print_dp(dp);
        cout << numbers[i] << " " << num_from_build << " " << num_from_build_alt << endl;
    }

    // example of a weird number, negative zero.
```

```
    double neg_zero{-0.0};

    cout << endl;
    cout << neg_zero << endl;

    print_dp(take_apart(neg_zero));

    return 0;
}
```

Here is the link to this program for downloading: double_parts.cpp

5.2 single_parts.cpp

Here is the shell of the program you will submit. Your job is to complete the missing definitions and functions.

```
// single_parts
#include <iostream>
#include <iomanip>

using namespace std;

// print out the parts of the structure Single_Parts
void print_sp(Single_Parts sp)
{
  if (sp.sign==1)
         cout << "negative"  << endl;
  else
        cout << "positive" << endl;

 cout << hex
      << setfill('0')
      << "expo: " << sp.exponent << endl
      << "frac: " << sp.fraction << endl
      << dec;
}

// define Single_Parts, build(), and take_apart() for float

int main()
{

    float num_from_build;

    float numbers[5]={1.0/3,2,1.3e10,3e11,6};
```

```
    // show the structure of the numbers
    for (int i=0;i<5;i++)
    {
        // take apart the numbers, then re-build to test that it works.

        Single_Parts s = take_apart(numbers[i]);
        num_from_build = build(s);

        cout << endl;
        print_sp(s);
        cout << numbers[i] << " " << num_from_build  << endl;
    }

    // example of a weird number, negative zero.
    double neg_zero{-0.0};

    cout << endl;
    cout << neg_zero << endl;

    print_sp(take_apart(neg_zero));

    return 0;
}
```

Here is the link the the starter program: single__parts.cpp

Note that the `main()` function of the program you submit must be identical to the one provided in `single_parts.cpp`. When the program is checked, whatever you have in `main()` will simply be replaced by the `main()` of `single_parts.cpp`

The filename of the program submitted must be `w2b_single_parts.cpp`

6 Part C: Half-precision adder

In this exercise, you will implement half-precision floating point. However, since neither language directly supports the format, what you will do is implement a python script that converts numbers in `half` format into python floats (which are actually double) and does the calculation there.

Think of `half` or `binary16` as a compact way to store floating point numbers, for example, for a high-resolution display.

The numbers are going to be given to you as 4 hexadecimal digits, representing the 16-bit `binary16` format.

Here is a shell for your program.

```
# w2c_addinghalf.py

from math import inf

def number_from_half(s : str):
    """return the number represented by s, a binary16 stored as a 4-character hex number"""
    return 0

def main():
    """add all binary16 numbers from standard input until a non-number is entered, then prin
    Numbers are represented in 4-character hex string format, one per line"""


if __name__ == '__main__':
    main()
```

You must implement both functions:

- `main()` which does the inputting from the terminal and adds the numbers together, printing the total.
- `number_from_half` which converts the string to a number.

The last part is boiler-plate python that allows this script to be usable either as a standalone program, like this:

`python w2c_addinghalf.py`

or as a library using import inside another script, like this:

`import w2c_addinghalf`

You can download the starter program here: adding_half.py

The filename of the program submitted must be `w2c_adding_half.py`

6.1 Hints

1. Use int(input(),16) to convert 4-character hex string to an integer, then manipulate it from there.
2. Use python

   ```
   try:
      #try something
   except:
     #handle error
   ```

to halt your program

6.2 Examples

You can test your program using these text files as example inputs:

This is test_ah_1.txt:

```
3c00
3f00
3c00
3e00
2c00
2cff
exit
```

This is test_ah_2.txt:

```
8123
0087
8102
exit
```

You can use the terminal to get this to be used as input using Unix “redirection operator” `<` as follows:

```
>python w2c_adding_half.py <test_ah_1.txt
5.39056396484375


>python w2c_adding_half.py <test_ah_2.txt
-2.467632293701172e-05
```