CS 137 Week 3

Floating Numbers, Math Library, Polynomials and Root Finding

September 25th, 2017

Floating Point Numbers

- How do we store decimal numbers in a computer?
- In scientific notation, we can represent numbers say by

$$-2.61202 \cdot 10^{30}$$

  where -2.61202 is called the precision and 30 is called the range.
- On a computer, we can do a similar thing to help store decimal numbers.

Data Types

Type	Size	Precision	Exponent
`float`	4 bytes	7 digits	± 38
`double`	8 bytes	16 digits	± 308

Note: You will almost always use the type `double`

Conversion Specifications

There are many different ways we can display these numbers using the `printf` command. They in general have the format `%`$\pm$ `m.pX` where

- $\pm$ is the right or left justification of the number depending on if the sign is positive or negative respectively
- `m` is the minimum field width, that is, how many spaces to leave for numbers
- `p` is the precision (this heavily depends on `X` as to what it means)
- `X` is a letter specifying the type (see next slide)

Conversion Specifications Continued

Some of the possible values for `X`

- `%d` refers to a decimal number. The precision here will refer to the minimum number of digits to display. Default is 1.
- `%e` refers to a float in exponential form. The precision here will refer to the number of digits to display after the decimal point. Default is 6.
- `%f` refers to a float in "fixed decimal" format. The precision here is the same as above.
- `%g` refers to a float in one of the two aforementioned forms depending on the number's size. The precision here is the maximum number of **significant digits** (not the number of decimal points!) to display. This is the most versatile option useful if you don't know the size of the number.

Example

```c
#include <stdio.h>
int main(void) {
  double x = -2.61202e30;
  printf("%zu\n",
    sizeof(double));
  printf("%f\n", x);
  printf("%.2e\n",x);
  printf("%g\n",x);
  return 0;
}
```

Notice that on the `%f` line above we get some garbage at the end (it is tough for a computer to store floating numbers!).

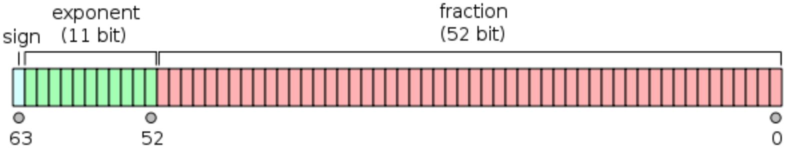
Exercise

Write the code that displays the following numbers (Ensure you get the white space correct as well!)

1. `3.14150e+10`
2. `  0436` (two leading white spaces)
3. `436   ` (three white spaces at the end)
4. `2.00001`

IEEE 754 Floating Point Standard

- IEEE - Institute of Electrical and Electronics Engineers



- Number is

$$(-1)^{\text{sign}} \cdot \text{fraction} \cdot 2^{\text{exponent}}$$

  (This is a bit of a lie but good enough for us - the details of this can get messy. See Wikipedia if you want more information)

(Picture courtesy of Wikipedia)

A Fun Aside

- How do I convert 0.1 as a decimal number to a decimal number in binary?
- Binary fractions are sometimes called 2-adic numbers.
- Idea: Write 0.1 as below where each a_i is one of 0 or 1 for all integers i.

$$0.1 = \frac{a_1}{2} + \frac{a_2}{4} + \frac{a_3}{8} + ... + \frac{a_k}{2^k} + ...$$

- Multiplying by 2 yields

$$0.2 = a_1 + \frac{a_2}{2} + \frac{a_3}{4} + ... + \frac{a_k}{2^{k-1}} + ...(\text{Eqn1})$$

  and so $a_1 = 0$ since $0.2 < 1$.
- Repeating gives

$$0.4 = a_2 + \frac{a_3}{2} + \frac{a_4}{4} + ... + \frac{a_k}{2^{k-2}} + ...$$

  and again $a_2 = 0$.

Continuing

- From

$$0.4 = 0 + \frac{a_3}{2} + \frac{a_4}{4} + ... + \frac{a_k}{2^{k-2}} + ...$$

multiplying by 2 gives

$$0.8 = a_3 + \frac{a_4}{2} + \frac{a_5}{4} ... + \frac{a_k}{2^{k-3}}$$

and again $a_3 = 0$. Doubling again gives

$$1.6 = a_4 + \frac{a_5}{2} + \frac{a_6}{4} ... + \frac{a_k}{2^{k-4}}$$

and so $a_4 = 1$. Now, we subtract 1 from both sides and then repeat to see that... (see next slide)

Continuing

$$1.6 - 1 = \frac{a_5}{2} + \frac{a_6}{4} ... + \frac{a_k}{2^{k-4}}$$
$$0.6 = \frac{a_5}{2} + \frac{a_6}{4} ... + \frac{a_k}{2^{k-4}}$$
$$1.2 = a_5 + \frac{a_6}{2} + \frac{a_7}{4} ... + \frac{a_k}{2^{k-4}}$$

giving $a_5 = 1$ as well. At this point, subtracting 1 from both sides gives

$$0.2 = \frac{a_6}{2} + \frac{a_7}{4} ... + \frac{a_k}{2^{k-4}}$$

which is the same as (Eqn 1) from two slides ago and hence,

$$(0.1)_{10} = (0.0\overline{0011})_2$$

Short Hand

$$
\begin{aligned}
0.1 \cdot 2 &= \mathbf{0}.2 \\
0.2 \cdot 2 &= \mathbf{0}.4 \\
0.4 \cdot 2 &= \mathbf{0}.8 \\
0.8 \cdot 2 &= \mathbf{1}.6 \\
0.6 \cdot 2 &= \mathbf{1}.2 \\
0.2 \cdot 2 &= \mathbf{0}.4
\end{aligned}
$$

and so $(0.1)_{10} = (0.0\overline{0011})_2$

Errors

- Notice that these floating point numbers only store rational numbers, that is, they cannot store real numbers (though there are CAS packages like Sage which try to).
- This for us is okay since the rationals can approximate real numbers as accurately as we need.
- When we discuss errors in approximation, we have two types of measures we commonly use, namely **absolute error** and **relative error**.

Errors (Continued)

- Let r be the real number we're approximating and let p be the exact value.
- Absolute Error $|p - r|$. Eg. $|3.14 - \pi| \approx 0.0015927...$
- Relative Error $\frac{|p-r|}{r}$. Eg. $\frac{|3.14-\pi|}{\pi} = 0.000507$.
- Note: Relative error can be large when r is small even if the absolute error is small.

Errors (Continued)

Be wary of...

- Subtracting nearly equal numbers
- Dividing by very small numbers
- Multiplying by very large numbers
- Testing for equality

An Example

```c
#include <stdio.h>
int main(void) {
  double a = 7.0/12.0;
  double b = 1.0/3.0;
  double c = 1.0/4.0;
  if (b+c==a) printf("Everything is Awesome!");
  else printf("Not cool... %g",b+c-a);
}
```

Watch out...

- Comparing $x == y$ is often risky.
- To be safe, instead of using `if (x==y)` you can use `if (x-y < 0.0001 && y-x < 0.0001)` (or use absolute values - see next lecture!)
- We sometimes call $\epsilon = 0.0001$ the **tolerance**.
- Note: Sometimes it is okay to compare floats to constants such as `if (x==0.0)` but you're best to exercise caution. Comparing to 0 is a surprisingly difficult problem.

One Note

- What happens when you type `double a = 1/3`? Do you get 0.33333?
- In C, most operators are overloaded. When it sees `1/3`, C reads this as integer division and so returns the value of 0.
- There are a few ways to fix this, one of them is to make at least one of the value a double (or a float) by writing `double a = 1.0/3` (dividing a double by an integer or a double gives a double).
- Another way is by **typecasting**, that is, explicitly telling C to make a value something else.
- For example, `double a = ((double)1)/3` will work as expected.

Math Library (Highlights)

- `#include <math.h>`
- Lots of interesting functions including:
  - `double sin(double x)` and similarly for `cos`, `tan`, `asin`, `acos`, `atan` etc.
  - `double exp(double x)` and similarly for `log`, `log10`, `log2`, `sqrt`, `fabs`, `ceil`, `floor` etc. (note log is the natural logarithm and `fabs` is the absolute value)
  - `int abs(int x)` is the absolute value function
  - `double pow(double x, double y)` gives x^y, the power function.
  - Constants: `M_PI`, `M_PI_2`, `M_PI_4`, `M_E`, `M_LN2`, `M_SQRT2`
  - Other values: `INFINITY`, `NAN`, `MAXFLOAT`

Polynomials

- A polynomial is an expression with at least one indeterminate and coefficients lying in some set.
- For example, $3x^3 + 4x^2 + 9x + 2$.
- In general: $p(x) = a_0 + a_1 x + ... + a_n x^n$
- We will primarily use ints for the coefficients. (maybe doubles later)
- Question: Brainstorm some different ways we can represent polynomials in memory. Discuss the pros and cons of each.

Our Representation

- We will represent it as an array of $n + 1$ coefficients where n is the degree.
- For our example $3x^3 + 4x^2 + 9x + 2$, we have `double p[] = {2.0, 9.0, 4.0, 3.0};`
- How do we evaluate a polynomial? That is, how can we implement:

```
double eval(double p[], int n, double x);
```

Traditional Method

- Compute x, x^2, x^3,.. x^n for $n-1$ multiplications.
- Multiply each by a_1, a_2, ..., a_n for another n multiplications.
- Add all the results $a_0 + a_1x + ... + a_nx^n$ for a final n multiplications.
- This gives a total of $2n-1$ multiplications and n additions.
- A note: Multiplication is an expensive operation compared to addition. Is there a way to reduce the number of multiplication operations?

Horner's Method

- Named after William George Horner (1786-1837) but known long before him (dating back as early as pre turn of millennium Chinese mathematicians).
- Idea:

$$2 + 9x + 4x^2 + 3x^3 = 2 + x(9 + x(4 + 3x))$$

- Start inside out. Total operations are n multiplications and n additions.

Horner's Method

```c
#include <stdio.h>
#include <assert.h>
double horner(double p[], int n, double x){
  assert(n > 0);
  double y = p[n-1];
  for(int i=n-2; i >= 0; i--)
    y = y*x + p[i];
  return y;
}
```

Horner's Method (Continued)

```
int main(void) {
  double p[] = {2,9,4,3};
  int len = sizeof(p)/sizeof(p[0]);
  printf("2 = %g\n",horner(p,len,0));
  printf("18 = %g\n",horner(p,len,1));
  printf("60 = %g\n",horner(p,len,2));
  printf("-6 = %g\n",horner(p,len,-1));
  return 0;
}
```

Root Finding

- Given a function $f(x)$, how can we determine a root?
- Example: $f(x) = x - \cos(x)$. Courtesy: Desmos.

Idea

- Notice that $f(-10) < 0 < f(10)$ so a root must be in the interval of $[-10, 10]$ (why!?)
- Look at the midpoint of the interval (namely 0) and evaluate $f(0)$.
- If $f(0) > 0$, look for a root in the interval $[-10, 0]$. Otherwise, look for a root in $[0, 10]$.
- Repeat until a root is found.

Bisection Method

- For which types of functions is this method guaranteed to work?
- What cases should we worry about?
- Can we run forever?
- What is our stopping condition?

Bisection Method

- For which types of functions is this method guaranteed to work?
- What cases should we worry about?
- Can we run forever?
- What is our stopping condition?
- Two stopping conditions possible
    - Stop when $|f(m)| < \epsilon$ for some fixed $\epsilon > 0$ where m is the midpoint of the interval.
    - Stop when $|m_{n-1} - m_n| < \epsilon$ (where m_n is the nth midpoint).
- Should include a safety escape, namely some fixed number of iterations.

Algorithm Pseudocode

- Given some a and b with $f(a) > 0$ and $f(b) < 0$, set $m = (a + b)/2$.
- If $f(m) < 0$, set $b = m$.
- Otherwise, set $a = m$
- Loop until either $|f(m)| < \epsilon$, $|m_{n-1} - m_n| < \epsilon$, or the number of iterations has been met.

Bisection.h

Bisection.h

```
#ifndef BISECTION_H
#define BISECTION_H
/*
Pre: None
Post: Returns the value of x - cos(x)
*/
double f(double x);
/*
Pre: epsilon > 0 is a tolerance, iterations > 0,
   f(x) has only one root in [a,b], f(a)f(b) < 0
Post: Returns an approximate root of f(x) using
   bisection method. Stops when either number of
   iterations is exceeded or |f(m)| < epsilon
*/
double bisect(double a, double b,
   double epsilon, int iterations);
#endif
```

Bisection.c

```c
#include <assert.h>
#include <math.h>
#include "bisection.h"
double f(double x){return x - cos(x);}
double bisect(double a, double b,
   double epsilon, int iterations){
  double m=a, fm;
  assert(epsilon > 0.0 && f(a)*f(b) < 0);
  for(int i=0; i<iterations; i++){
    m = (a+b)/2.0;
    fm = f(m); //Why is this a good idea?
    if (fabs(fm) < epsilon) return m;
    if (fm*f(b) > 0) b=m;
    else a=m;
  }
  return m;
}
```

Main.c

```c
#include <stdio.h>
#include "bisection.h"
int main(void) {
  printf("%g\n", bisect(-10,10,0.0001,50));
  return 0;
}
```

Calculating the Number of Iterations

- We used the break condition $|f(m)| < \epsilon$ in our code.
- One advantage to using the condition $|m_n - m_{n-1}| < \epsilon$ is that we can compute the number of iterations fairly easily.
- After each iteration, the length of the interval is cut in half, so, we seek to find a value for n such that

$$\epsilon > \frac{b-a}{2^n}$$

rearranging gives

$$2^n > \frac{b-a}{\epsilon}$$

and so after logarithms

$$n \log 2 > \log(b-a) - \log(\epsilon)$$

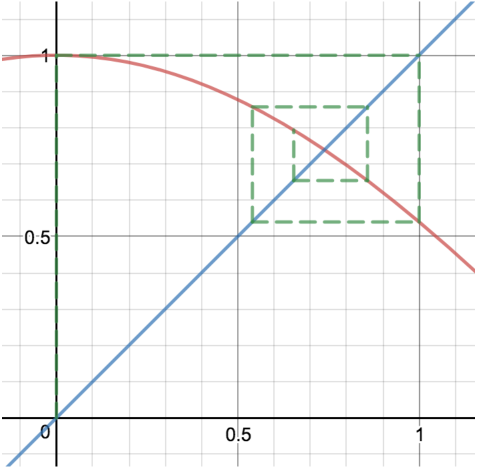
with $b = 10$, $a = -10$, $\epsilon = 0.0001$, we get $n > 17.60964$.

Another Method - Fixed Point Iteration

- Given a function $g(x)$, we seek to find a value x_0 such that $g(x_0) = x_0$.
- We call such a point a fixed point.
- These are of significant importance in dynamical systems.
- In our example, looking for a root of $f(x) = x - \cos(x)$ is the same problem as finding a fixed point of $g(x) = \cos(x)$.
- Note: Not all functions have fixed points (but we can transfer between root solving problems and fixed point problems).
- There is another more visual way to interpret this...

Cobwebbing

Also known as **Cobwebbing**. (Courtesy Desmos)



A Note

$$
\begin{aligned}
x_0 &= 0 \\
g(x_0) &= 1 \\
g(g(x_0)) &= g(1) = 0.540 \\
g(g(g(x_0))) &= g(g(1)) = g(0.540) = 0.858 \\
g(g(g(g(x_0)))) &= g(g(g(1))) = g(g(0.540)) = g(0.858) = 0.654
\end{aligned}
$$

- It turns out by the Banach Contraction Mapping Theorem (or the Banach Fixed Point Theorem) that if the slope of the tangent line at a fixed point has magnitude less than 1, this cobwebbing process will eventually converge to a suitable starting point.

Pseudocode

- Start with some point x_0.
- Compute $x_1 = g(x_0)$.
- If $|x_1 - x_0| < \epsilon$, stop.
- Otherwise go back to the beginning with $x_0 = x_1$.

Fixed.h

Fixed.h

```
#ifndef FIXED_H
#define FIXED_H
/* Pre: None
   Post: Returns the value of cos(x) */
double g(double x);
/*
Pre: epsilon > 0 is a tolerance, iterations > 0,
 x0 is sufficiently close to a stable fixed point
Post: Returns an approximate fixed point of g(x)
 using cobwebbing. Stops when either number of
 iterations is exceeded or |g(xi)-xi| < epsilon
 where xi is the value of x0 after i iterations.
*/
double fixed(double x0, double epsilon,
   int iterations);
#endif
```

Fixed.c

```c
#include <assert.h>
#include <math.h>
#include "fixed.h"
double g(double x){return cos(x);}
double fixed(double x0,
   double epsilon, int iterations){
  double x1;
  assert(epsilon > 0.0);
  for(int i=0; i<iterations; i++){
    x1 = g(x0);
    if (fabs(x1-x0) < epsilon) return x1;
    x0 = x1;
  }
  return x0;
}
```

Main.c

```c
#include <stdio.h>
#include "fixed.h"
int main(void) {
  printf("%g\n", fixed(0,0.0001,50));
  return 0;
}
```

Improving the previous two codes

- Notice in each of the two previous examples, we hard coded a definition of a function.
- Ideally, the code would also have as a parameter the function itself.
- C lets us do this using function pointers.
- Syntax: Pass a parameter `double (*f)(double)` a pointer to a function that consumes a double and returns a double.
- Note: The brackets around `(*f)` are important to not confuse this with a function that returns a pointer.

Bisection2.h

Bisection2.h

```
#ifndef BISECTION2_H
#define BISECTION2_H
double bisect2(double a, double b,
   double epsilon, int iterations,
   double (*f)(double));
#endif
```

Bisection2.c

```
#include <assert.h>
#include <math.h>
#include "bisection2.h"
double bisect2(double a, double b,
    double epsilon, int iterations,
    double (*f)(double)){
  double m=a, fm;
  assert(epsilon > 0.0 && f(a)*f(b) < 0);
  for(int i=0; i<iterations; i++){
    m = (a+b)/2.0;
    fm = f(m);
    if (fabs(fm) < epsilon) return m;
    if (fm*f(b) > 0) b=m;
    else a=m;
  }
  return m;
}
```

Main.c

```c
#include <stdio.h>
#include <math.h>
#include "bisection2.h"

double g(double x){return x - cos(x);}
double h(double x){return x*x*x-x+1;}

int main(void) {
  printf("%g\n", bisect2(-10,10,0.0001,50,g));
  printf("%g\n", bisect2(-10,10,0.0001,50,h));
  return 0;
}
```