CS 137

Floating Numbers, Math Library, Polynomials, and Root Finding

Fall 2025

Victoria Sakhnini

Table of Contents

Floating Point Numbers

Let us explore how to work with decimal numbers. You can write decimal numbers using decimal points such as 3.125 or scientific notation such as $-2.61202 \text{ x } 10^{30}$, where -2.61202 is the precision, and 30 is the range. In C we will use two data types:

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

You will almost always use the type double.

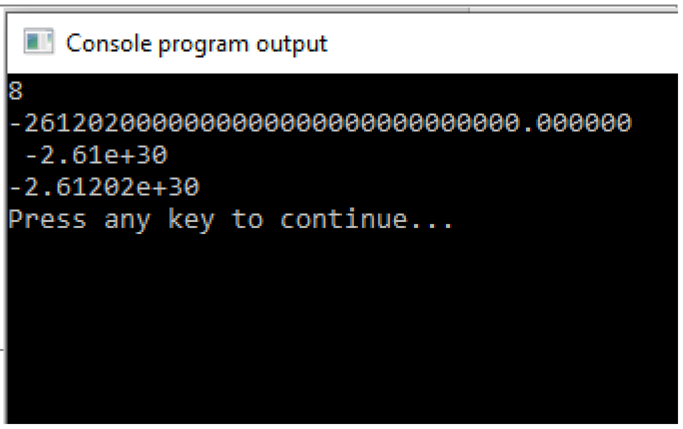
printf and decimal numbers

We can display these numbers using the printf command in many different ways. They, in general, have the format %± m.pX where

- ± is the right or left justification of the number depending on whether the sign is positive or negative, and
- 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. Some possible values for X:
    - %d refers to a decimal number. The precision here will refer to the minimum number of digits to display. The 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. The 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 is the maximum number of significant digits (not the number of decimal points!) to display. This is the most versatile option, and it is useful if you don't know the size of the number.

Example:

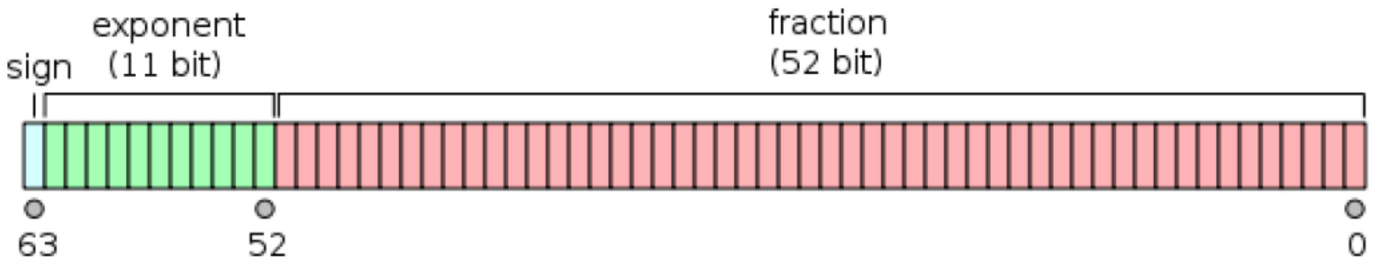
```c
#include <stdio.h>

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;
}
```



```
8
-2612020000000000000000000000000.000000
 -2.61e+30
-2.61202e+30
Press any key to continue...
```

IEEE 754 Floating Point Standard

IEEE - Institute of Electrical and Electronics Engineers



(Picture courtesy of Wikipedia)

Number is[1] $(-1)^{sign} \times \text{fraction} \times 2^{exponent}$

Converting decimal to Binary

1. To convert the fractional part to Binary, multiply the fractional part by 2 and take the one bit that appears before the decimal point.
2. Follow the same procedure after the decimal point (.) part until it becomes 1.0.

Example1 (`0.25`):

`0.25 * 2 =0.50` //take `0` and move `0.50` to the next step

`0.50 * 2 =1.00` //take `1` and stop the process

Thus: $0.25 = (0.01)_2$

[1] (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)

Example2 (0.33):

```
0.33 * 2 =>0.66          // take 0  and move  .66 to next step
0.66 * 2 =>1.32          // take 1 and move .32 to next step
0.32 * 2 =>0.64          // take 0  and move .64 to next step
0.64 * 2 =>1.28          // take 1 and move .28 to next step
0.28 * 2 =>0.56          // take 0 and move .56 to next step
0.56 * 2 =>1.12          // take 1 and move .12 to next step
0.12 * 2 =>0.24          // take 0 and move .24 to next step
0.24 * 2 =>0.48          // take 0 and move .48 to next step
0.48 * 2 =>0.96          // take 0 and move .96 to next step
0.96 * 2 =>1.92          // take 1 and move .92 to next step
0.92 * 2 =>1.84          // take 1 and move .84 to next step
0.84 * 2 =>1.68          // take 1 and move .68 to next step
0.68 * 2 =>1.36          // take 0 and move .36 to next step
0.36 * 2 =>0.72          // take 0 and move .72 to next step
0.72 * 2 =>1.44          // take 1 and move .44 to next step
0.44 * 2 =>0.88          // take 0 and move .88 to next step
0.88 * 2 =>1.76          // take 1 and move .76 to next step
0.76 * 2 =>1.32          // take 1 and move .32 to next step
```

Here, the fractional part 0.32 is repeated.

With the above method, some fractional part numbers will not end up with 1.0.

The computer will allocate 23 bits for the fractional part in floating number storage. So, it's enough to do the above method 23 times.

$0.33 = (0.010101000111001011)_2$

Errors

Notice that these floating-point numbers only store rational numbers; they cannot store real numbers (though there are CAS packages like Sage that try to). This is okay since rational numbers can approximate real numbers as accurately as we need.

When discussing approximation errors, we commonly use two types of measures: absolute and relative.

Let r be the real number we're approximating, and p be the approximate value.

$$\text{Absolute Error } |p - r|. \text{ Eg. } |3.14 - \pi| \approx 0.0015927\ldots$$

$$\text{Relative Error } \frac{|p-r|}{|r|}. \text{ Eg. } \frac{|3.14-\pi|}{|\pi|} = 0.000507.$$

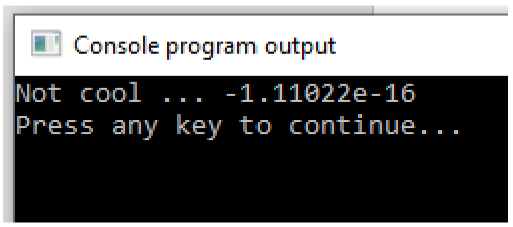
Note: Relative error can be significant when r is small, even if the absolute error is small.

Be wary of...

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

Example1:

```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\n", b + c - a);
 }
```



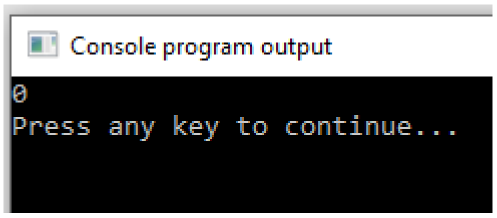
```
Not cool ... -1.11022e-16
Press any key to continue...
```

Watch out for:

- 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)
- We sometimes call ε `= 0.0001` the tolerance.

Example2: Consider the following program and output:

```c
#include <stdio.h>

int main(void)
{
  double a = 1 / 3;
  printf("%g\n", a);
  return 0;
}
```

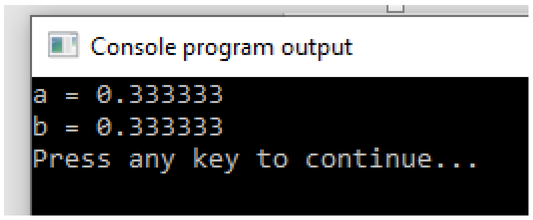


```
0
Press any key to continue...
```

Do we get `0.33333`? NO !!!!!

In `C`, most operators are overloaded. When it sees `1/3`, C reads this as an integer division and returns the value of `0`. There are a few ways to fix this: to make at least one of the values 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, explicitly telling `C` to make a value something else. For example, `double a = (double)1/3` will work as expected.

```c
#include <stdio.h>

int main(void)
{
        double a = 1.0 / 3;
        double b = (double)1 / 3;
        printf("a = %g\n", a);
        printf("b = %g\n", b);
        return 0;
 }
```



```
a = 0.333333
b = 0.333333
Press any key to continue...
```

Math Library

http://www.tutorialspoint.com/c_standard_library/math_h.htm

```
#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`, `ceil`, `floor` etc. (note `log` is the natural logarithm and `fabs` is the absolute value)
- `double fabs(double x)` is the absolute value function for floats (integer `abs` is in `stdlib`)
- `double pow(double x, double y)` gives x^y , the power function.
- Constants (Caution! These following two lines are not in the basic standard!):
  `M_PI` (Pi), `M_PI_2` (Pi divided by 2), `M_PI_4` (Pi divided by 4), `M_E`, `M_LN2`, `M_SQRT2` (The square root of 2)
- Other values: `INFINITY`, `NAN` (square root of -1), `MAXFLOAT`

We will see some examples later using this library; you can try the above functions with simple programs to explore them. In this course, we will primarily work with integers in future chapters.

Note [IMPORTANT]:

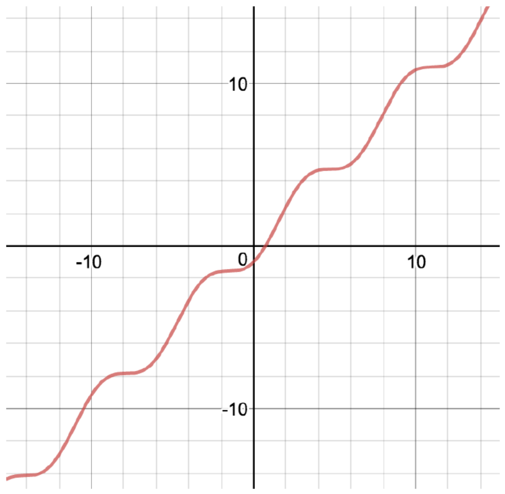
To compile a C program that includes math.h library in Linux, you must add -lm at the end of the command line. For example:

```
gcc  -std=c11 program.c -lm
```

Root Finding

Given a function `f(x)`, how can we determine a root? Finding the root involves finding the value of `x` for which `f(x)=0`.

Example: `f(x) = x - cos(x)`.   Courtesy: Desmos.



Claim: When `f(x) = x - cos(x)`, there is a root in the interval `[-10, 10]`.

Idea:

- Notice that `f(-10) < 0 < f(10)` so a root must be in the interval of `[-10, 10]`.
- Look at the interval's midpoint (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.

Method 1: Bisection Method

It is based on the theory that if a function `f(x)` is continuous on an interval [`a,b`] and `f(a)*f(b)<0`, [2] then a value `c` in `(a,b)` exists for which `f(c)=0`;

Following the idea above, the algorithm ends when the value of `f(c)` is less than a defined tolerance. In this case, we say that `c` is close enough to be the root of the function. We can set a maximum number of iterations to avoid too many iterations.

There are two stopping conditions possible:

- Stop when $|f(m)|<\varepsilon$ for some fixed $\varepsilon>0$ where `m` is the interval's midpoint. (Not great since the actual root might still be far away)
- Stop when $|m_{n-1} - m_n|<\varepsilon$ (where m_n is the n^{th} midpoint). (Much better)

[2] for `f(a)*f(b)` to be negative it means that `f(a)` is positive and `f(b)` is negative or vice versa.

Algorithm:

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 $|f(m)|< \varepsilon$, $|m_{n-1} - m_n|<\varepsilon$, or the number of iterations has been met.

Implementation:

Interface file

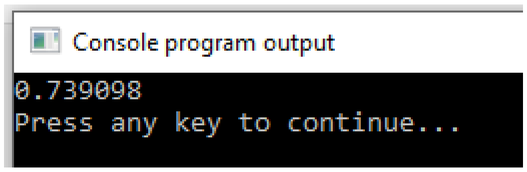
```
#ifndef BISECTION_H
#define BISECTION_H

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
```

Implementation file

```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;
         double fb = f(b); // Why is this a good idea?
         assert(epsilon > 0.0 && f(a)*f(b) < 0);

         for(int i=0; i< iterations ; i ++){
                 m = (a+b )/2.0;
                 if (fabs (b-a) < epsilon){
                         return m;
                 }
                 // Alternatively :
                 // if ( fabs (f(m)) < epsilon ) return m;
                 if (f(m)* fb > 0) {
                         b = m;
                         fb = f(b);
                 }
                 else {
                         a=m;
                 }
         }
         return m;
 }
```

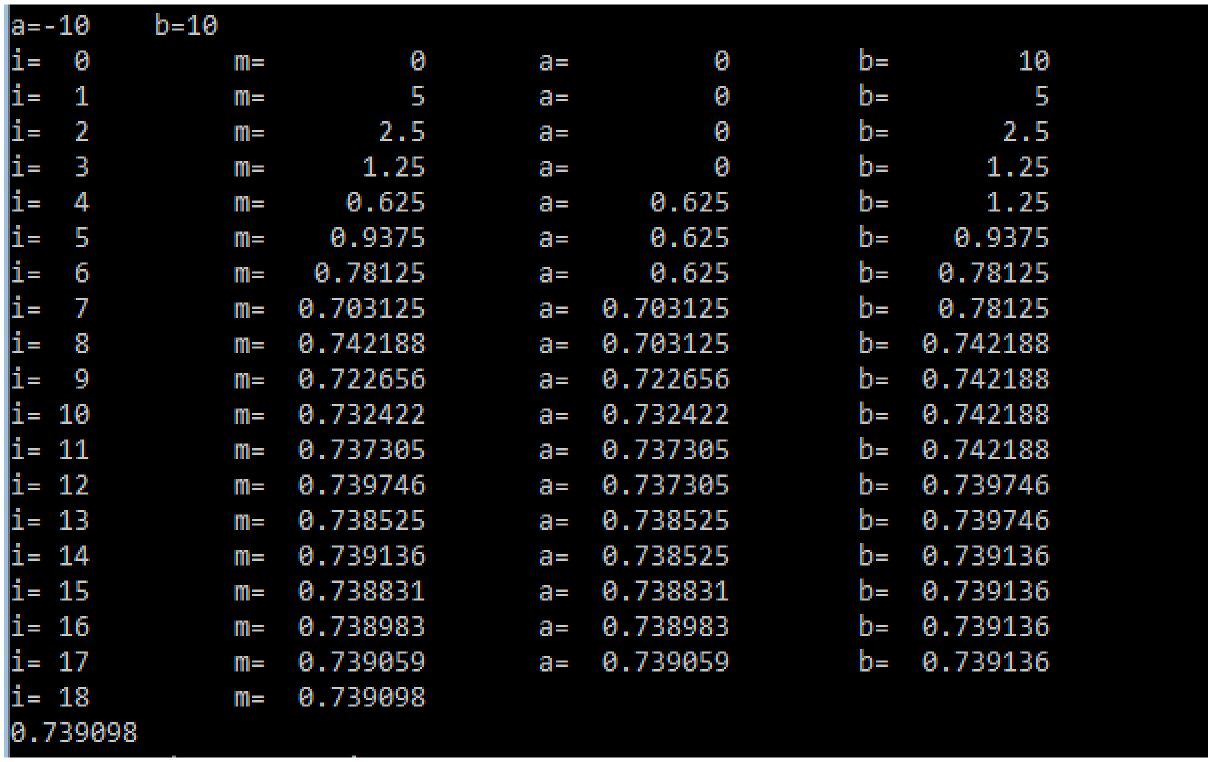
Program

```c
#include <stdio.h>
#include "bisection.h"

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


Console program output

```
0.739098
Press any key to continue...
```

Tracing:



```
a=-10    b=10
i=  0         m=         0       a=         0         b=        10
i=  1         m=         5       a=         0         b=         5
i=  2         m=       2.5       a=         0         b=       2.5
i=  3         m=      1.25       a=         0         b=      1.25
i=  4         m=     0.625       a=     0.625         b=      1.25
i=  5         m=    0.9375       a=     0.625         b=    0.9375
i=  6         m=   0.78125       a=     0.625         b=   0.78125
i=  7         m=  0.703125       a=  0.703125         b=   0.78125
i=  8         m=  0.742188       a=  0.703125         b=  0.742188
i=  9         m=  0.722656       a=  0.722656         b=  0.742188
i= 10         m=  0.732422       a=  0.732422         b=  0.742188
i= 11         m=  0.737305       a=  0.737305         b=  0.742188
i= 12         m=  0.739746       a=  0.737305         b=  0.739746
i= 13         m=  0.738525       a=  0.738525         b=  0.739746
i= 14         m=  0.739136       a=  0.738525         b=  0.739136
i= 15         m=  0.738831       a=  0.738831         b=  0.739136
i= 16         m=  0.738983       a=  0.738983         b=  0.739136
i= 17         m=  0.739059       a=  0.739059         b=  0.739136
i= 18         m=  0.739098
0.739098
```

Calculating the number of iterations

An advantage to using the condition $|m_n - m_{n-1}| < \epsilon$ is that this gives us good accuracy on the actual root.

Another is that we can compute the number of iterations fairly easily (and so don't necessarily need our iterations guard).

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 $n > 17.60964$.

Method 2: Fixed Point Iteration

Goal: Given a function $g(x)$, we seek to find a value x_0 such that $g(x_0)=x_0$.

x_0 is called a fixed point. This topic is important in dynamic systems (You will learn about this in future courses). 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 and fixed-point problems).

Algorithm:

Start with some point x_0.

Compute $x_1 = g(x_0)$.

If $|x_1 - x_0|<\varepsilon$, stop.

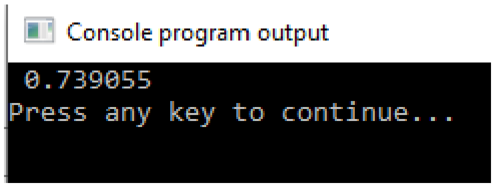
Otherwise, go back to the beginning with $x_0 = x_1$.

Implementation:

```c
#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
```

```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;
}
```

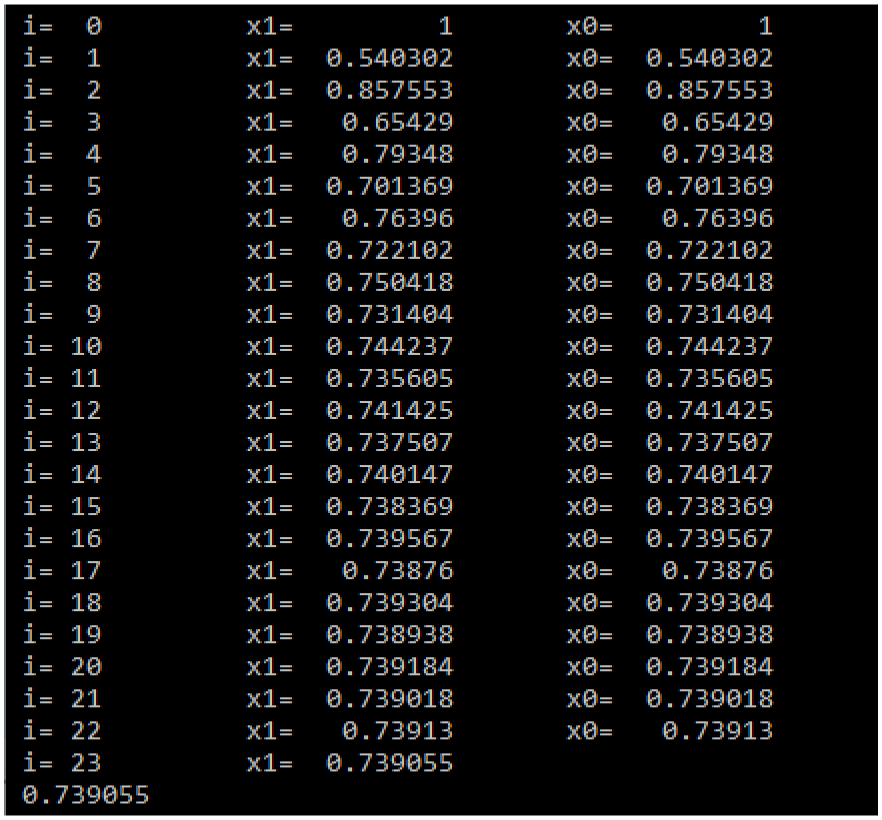
```c
#include <stdio.h>
#include "fixed.h"

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



Console program output

```
 0.739055
Press any key to continue...
```

Tracing:



```
i=  0        x1=        1        x0=        1
i=  1        x1= 0.540302        x0= 0.540302
i=  2        x1= 0.857553        x0= 0.857553
i=  3        x1=  0.65429        x0=  0.65429
i=  4        x1=  0.79348        x0=  0.79348
i=  5        x1= 0.701369        x0= 0.701369
i=  6        x1=  0.76396        x0=  0.76396
i=  7        x1= 0.722102        x0= 0.722102
i=  8        x1= 0.750418        x0= 0.750418
i=  9        x1= 0.731404        x0= 0.731404
i= 10        x1= 0.744237        x0= 0.744237
i= 11        x1= 0.735605        x0= 0.735605
i= 12        x1= 0.741425        x0= 0.741425
i= 13        x1= 0.737507        x0= 0.737507
i= 14        x1= 0.740147        x0= 0.740147
i= 15        x1= 0.738369        x0= 0.738369
i= 16        x1= 0.739567        x0= 0.739567
i= 17        x1=  0.73876        x0=  0.73876
i= 18        x1= 0.739304        x0= 0.739304
i= 19        x1= 0.738938        x0= 0.738938
i= 20        x1= 0.739184        x0= 0.739184
i= 21        x1= 0.739018        x0= 0.739018
i= 22        x1=  0.73913        x0=  0.73913
i= 23        x1= 0.739055
0.739055
```

function pointer

Notice that we hard-coded a function definition in the two previous examples. Ideally, the code would also have the function itself as a parameter. C lets us do these using function pointers.

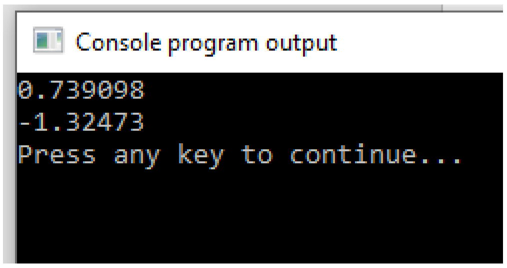
Syntax: Pass a parameter `double (*f)(double)` a pointer to a function that takes a double and returns a double.

Note: The brackets around `(*f)` are important to avoid confusing this with a function that returns a pointer. In our function, we pass a pointer to another function, which takes a double as input and returns a double.

```c
#ifndef BISECTION2_H
#define BISECTION2_H
#include <math.h>
/*
   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 bisect2(double a, double b, double epsilon, int iterations,
         double (*f) (double));

 # endif
```

```c
#include <assert.h>
#include "bisection2.h"

 // Note: no need to include math.h as it was included in the interface file

double bisect2(double a, double b, double epsilon, int iterations, double (*f)
(double))
{
    double m = a;
    double fb = f(b);
          assert(epsilon > 0.0 && f(a) * f(b) < 0);
    for (int i = 0; i < iterations; i++)
    {
          m = (a + b) / 2.0;
          if (fabs(b - a) < epsilon)
                return m;
          // Alternatively :
          // if ( fabs (f(m)) < epsilon ) return m;
          if (f(m) * fb > 0)
          {
                b = m;
                fb = f(b);
          }
          else
          {
                a = m;
          }
    }
    return m;
 }
```

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

double f1(double x)
{
  return x - cos(x);
}

double f2(double x)
 {
         return x * x * x - x + 1;
 }

 int main(void)
 {
         // Note how we called bisect2 twice, each time with
         // a different function (f1, then f2)

         printf("%g\n", bisect2(-10, 10, 0.0001, 50, f1));
         printf("%g\n", bisect2(-10, 10, 0.0001, 50, f2));
         return 0;
 }
```


Console program output

```
0.739098
-1.32473
Press any key to continue...
```

Polynomials

A polynomial is an expression with at least one indeterminate and coefficients lying in some set. In general: $p(x) = a_0 + a_1x + \ldots + a_nx^n$. For example, $3x^3 + 4x^2 + 9x + 2$.

We will primarily use integers for the coefficients. (maybe doubles later).

Think about some different ways we can represent polynomials in memory. Then, consider the pros and cons of each before you continue reading.

This Photo by Unknown Author is licensed under CC BY-ND

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 $x, x^2, x^3, .. x^n$ ($n-1$ multiplications). Multiply each by $a_1, a_2, ..., a_n$ (another n multiplications). Add all the results $a_0 + a_1x + …. + a_nx^n$ (n additions). This gives a total of $2n-1$ multiplications and n additions.

Multiplication is an expensive operation (computationally) compared to addition.

Is there a way to reduce the number of multiplication operations?

Horner's Method

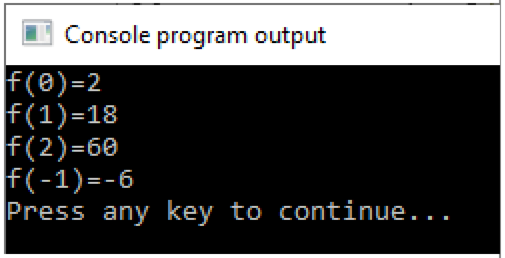
Idea: $a_0 + a_1x + …. + a_nx^n$ is equivalent to $a_0 + x(a_1 +x(a_2 + …… x(a_{n-1}+ a_nx)…)$

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

Total operations = n multiplications + n additions (compared to traditional evaluation, which requires $(2n-1)$ multiplications and n additions.

Implementation:

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

double horner(double p[], int n, double x){
    assert(n > 0);
    // n is the number of elements in the array thus the polynomial has degree n-1
    double y = p[n - 1];
    for (int i = n - 2; i >= 0; i--)
          y = y * x + p[i];
    return y;
 }

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


Console program output

```
f(0)=2
f(1)=18
f(2)=60
f(-1)=-6
Press any key to continue...
```

Extra Practice Problems

1) Write a function named `find_sqrt` that implements Newton's Iteration algorithm to compute the square root of `n`.

The function has two `double` parameters `n` and `tolerance`, and returns the square root as a `double`.

Starting with an initial guess of $x_0 = 1$, each subsequent guess x_{k+1} is calculated as:
$x_{k+1} = (x_k + n/x_k)/2$

Your function must stop and return `x` when the absolute difference between x^2 and `n` is less than or equal to `tolerance`.

The solution is provided[i]. My advice is to solve the question before you look at my solution.

Here is a `main` function that you can use to test your solution:

```c
int main(void)
{

  assert(find_sqrt(100.500000, 0.01) * find_sqrt(100.500000, 0.01) - 100.500000 <= 0.01);
  assert(find_sqrt(1123.525, 0.00123) * find_sqrt(1123.525, 0.00123) - 1123.525 <= 0.00123);
  assert(find_sqrt(10000.250000, 0.00000000001) * find_sqrt(10000.250000, 0.00000000001) - 10000.250000 <= 0.00000000001);
}
```

2) What is the output[ii] of the following program (Trace manually before you run it on a computer).

```c
#include <stdio.h>
int main(void)
{
          int i = 1, g = 2, *pi, *ptr = &i;
          float f, *pf;
          char c = 0, *pc;
          pi = &g;
          pc = &c;
          pf = &f;
          *pi += 43;
          *pf = (float)*pi + 0.54;
          ptr -= (int)*pf;
          *pc += '0';
          printf("%c\n",c);
          printf("%d\n",c);
          *pc += 4;
          printf("%d,%d\n", i, g);
          printf("%.2f\n", f);
          printf("%c\n", c);
          printf("%d\n",c);
          return 0;
  }
```

3) Write a program to compute the cosine of x. The user should supply x and a positive integer n. We compute the cosine of x using the series and the computation should use all terms in the series up through the term involving x^n

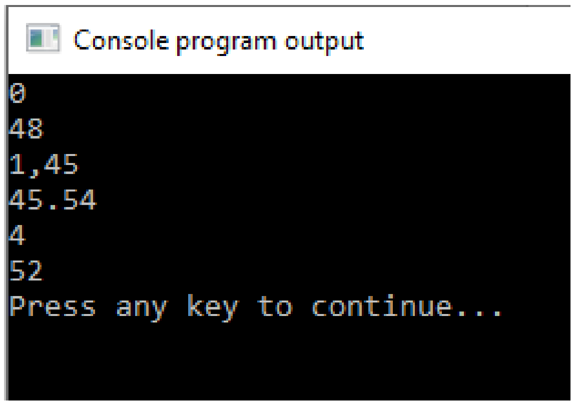
$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! \ldots..$

Answers

i

```
#include <assert.h>
#include <stdbool.h>

// within_tolerance(a, b, tolerance) returns true if the difference between
//   a and b is <= tolerance, otherwise false.
// requires: tolerance >= 0
bool within_tolerance(double a, double b, double tolerance)
{
    if (a < b)
    {
        return (b - a <= tolerance);
    }
    else
    {
        return (a - b <= tolerance);
    }
}

// find_sqrt(n, tolerance) uses Newton's Iteration algorithm to find the
//   sqrt of n within the given tolerance
// requires: n > 0, tolerance > 0
double find_sqrt(double n, double tolerance)
{
    double g = 1;

    while (!within_tolerance(n, g * g, tolerance))
    {
        g = (g + n / g) / 2;
    }
    return g;
}

int main(void)
{

    assert(find_sqrt(100.500000, 0.01) * find_sqrt(100.500000, 0.01)
        - 100.500000 <= 0.01);
    assert(find_sqrt(1123.525, 0.00123) * find_sqrt(1123.525, 0.00123)
        - 1123.525 <= 0.00123);
    assert(find_sqrt(10000.250000, 0.00000000001) * find_sqrt(10000.250000, 0.00000000001)
        - 10000.250000 <= 0.00000000001);
}
```

ii



Console program output

```
0
48
1,45
45.54
4
52
Press any key to continue...
```