CS 137

Programming Principles

Chapter 7

Floating Numbers, Math Library, Polynomials, and Root Finding

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 in scientific notation such as `-2.61202 × 10^30`, where `-2.61202` is the precision and `30` is the range. In C we will use two data types:

Type	Size	Precision	Range (approx.)
float	4 bytes	~7 significant digits	$\pm 3.4 \times 10^{38}$
double	8 bytes	~15 significant digits	$\pm 1.7 \times 10^{308}$

> You will almost always use the type double.

printf and Decimal Numbers

We can display floating-point numbers using `printf` in many different ways. The general format is `%± m.pX` where:

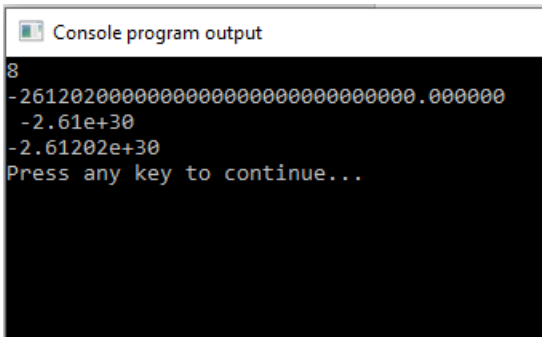
- `±` is the right or left justification (positive = right, negative = left).
- `m` is the minimum field width — how many spaces to reserve for the number.
- `p` is the precision (meaning depends on X).
- `X` is a letter specifying the type:

- `%d` — decimal integer. Precision = minimum number of digits to display (default 1).
- `%e` — float in exponential form. Precision = digits after the decimal point (default 6).
- `%f` — float in fixed decimal format. Precision same as %e.
- `%g` — float in either %e or %f form depending on the number's size. Precision = maximum number of significant digits. Most versatile; useful when the magnitude of the number is unknown.

Example

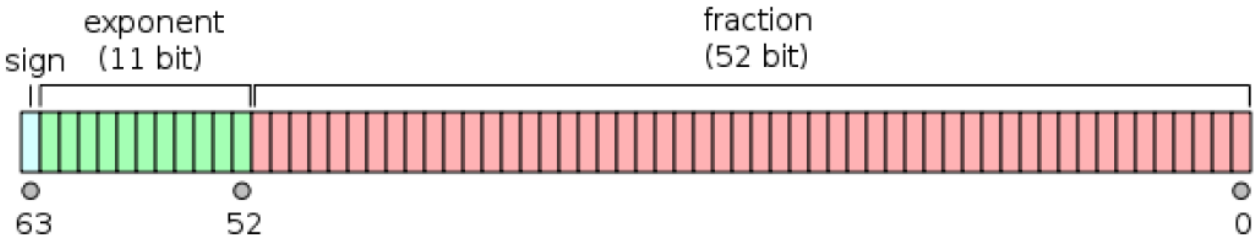
```
#include <stdio.h>

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;
}
```



IEEE 754 Floating Point Standard

IEEE — Institute of Electrical and Electronics Engineers.



(Picture courtesy of Wikipedia)

A floating-point number is stored as:

```
(-1)^(sign)  ×  fraction  ×  2^(exponent)
```

The sign bit, exponent, and mantissa (fraction) are stored in separate bit fields. A single-precision (`float`) uses 32 bits; a double-precision (`double`) uses 64 bits.

> 📌 The formula above is a simplification that is good enough for our purposes. The full details can get complex — see the Wikipedia article on IEEE 754 for more information.

Converting Decimal to Binary

To convert the fractional part to binary:

- Multiply the fractional part by 2 and record the bit that appears before the decimal point.
- Repeat with the new fractional part until you reach 1.0 or have enough bits.

Example 1 — 0.25

```
0.25 × 2 = 0.50  →  take 0,  carry 0.50
0.50 × 2 = 1.00  →  take 1,  stop

Result: 0.25 = (0.01)₂
```

Example 2 — 0.33

```
0.33 × 2 = 0.66  →  0
0.66 × 2 = 1.32  →  1   carry 0.32
0.32 × 2 = 0.64  →  0
0.64 × 2 = 1.28  →  1   carry 0.28
0.28 × 2 = 0.56  →  0
0.56 × 2 = 1.12  →  1   carry 0.12
0.12 × 2 = 0.24  →  0
0.24 × 2 = 0.48  →  0
0.48 × 2 = 0.96  →  0
0.96 × 2 = 1.92  →  1   carry 0.92
0.92 × 2 = 1.84  →  1   carry 0.84
0.84 × 2 = 1.68  →  1   carry 0.68
0.68 × 2 = 1.36  →  1   carry 0.36   ← note: 0.36 will repeat
0.36 × 2 = 0.72  →  0
0.72 × 2 = 1.44  →  1   carry 0.44
0.44 × 2 = 0.88  →  0
0.88 × 2 = 1.76  →  1   carry 0.76
0.76 × 2 = 1.52  →  1   carry 0.52
          ... 0.32 repeats — the pattern is periodic

0.33 ≈ (0.01010100011100101 1)₂   (23 bits stored by the hardware)
```

📌 Some fractional numbers never terminate in binary (just as 1/3 never terminates in decimal). The computer allocates 23 bits for the fractional part of a float, so the conversion stops after 23 iterations.

Errors

Floating-point numbers can only store rational numbers; they cannot store arbitrary real numbers exactly. Rational numbers can approximate real numbers as accurately as we need, but approximation errors exist.

Let r be the real number being approximated, and p be the approximate value. We use two error measures:

```
Absolute error  =  |p - r|
Relative error  =  |p - r| / |r|    (when r ≠ 0)
```

⚠ Relative error can be significant when r is small, even if the absolute error appears small.

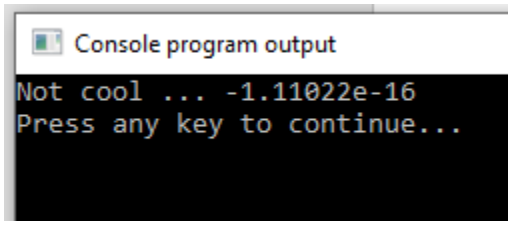
Be wary of:

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

Example 1 — Equality comparison pitfall

```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);
}
```



⚠ Comparing x == y is often risky with floating-point numbers. Instead of if (x == y), use a tolerance: if (x - y < 0.0001 || y - x < 0.0001) or equivalently: if (fabs(x - y) < 0.0001) The value ε = 0.0001 is called the tolerance.

Example 2 — Integer division pitfall

```c
#include <stdio.h>

int main(void)
{
    double a = 1 / 3;    // integer division! result is 0, not 0.333...
    printf("%g\n", a);
    return 0;
}
```

In C, most operators are overloaded. When C sees `1/3`, it reads this as integer division and returns 0. To fix this, make at least one operand a `double`:

```c
#include <stdio.h>

int main(void)
{
    double a = 1.0 / 3;        // double literal — works correctly
    double b = (double)1 / 3;  // explicit typecast — also works
    printf("a = %g\n", a);
    printf("b = %g\n", b);
    return 0;
}
```

Math Library

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

```c
#include <math.h>
```

Useful functions include:

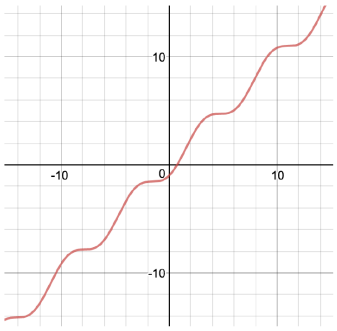
- `double sin(double x)` — and similarly `cos`, `tan`, `asin`, `acos`, `atan`, etc.
- `double exp(double x)` — and similarly `log` (natural log), `log10`, `log2`, `sqrt`, `ceil`, `floor`, etc.
- `double fabs(double x)` — absolute value for floating-point (integer `abs` is in `<stdlib.h>`).
- `double pow(double x, double y)` — computes x^y.
- Constants (caution — not in the basic standard): `M_PI`, `M_PI_2`, `M_PI_4`, `M_E`, `M_LN2`, `M_SQRT2`.
- Other special values: `INFINITY`, `NAN` (e.g., √−1), `MAXFLOAT`.

⚠ IMPORTANT: To compile a C program that includes <math.h> on Linux, you must add -lm at the end of the compile command: gcc -std=c11 program.c -lm

Root Finding

Given a function f(x), how can we find a root — that is, a value of x for which f(x) = 0?

Example: f(x) = x − cos(x).



Claim: There is a root in the interval [−10, 10].

Idea:

- Notice that f(−10) < 0 < f(10), so a root must lie in [−10, 10].
- Look at the midpoint (0) and evaluate f(0).
- If f(0) > 0, search in [−10, 0]. Otherwise, search in [0, 10].
- Repeat until a root is found.

Method 1: Bisection Method

Based on the theorem: if f(x) is continuous on [a, b] and f(a) × f(b) < 0, then there exists c in (a, b) such that f(c) = 0.

> 📌 f(a) × f(b) < 0 means f(a) and f(b) have opposite signs.

The algorithm ends when |f(c)| is less than a defined tolerance ε, or a maximum number of iterations is reached.

Two stopping conditions:

- Stop when |f(m)| < ε for some fixed ε > 0, where m is the interval midpoint. (Not great — the actual root may still be far away.)
- Stop when |m_(n−1) − m_(n)| < ε, where m_(n) is the nth midpoint. (Much better.)

Algorithm

```
Given a and b with f(a) and f(b) of opposite sign, set m = (a + b) / 2.

If f(m) and f(b) have the same sign, set b = m.
Otherwise, set a = m.

Loop until |f(m)| < ε, |m_(n-1) - m_(n)| < ε, or max iterations reached.
```

Interface File — bisection.h

```
#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 the bisection method.
 *       Stops when either the number of iterations is exceeded
 *       or |f(m)| < epsilon.
 */
double bisect(double a, double b, double epsilon, int iterations);

#endif
```

Implementation File — bisection.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);  // cached to avoid recomputing f(b) each iteration
    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;
}
```

Main Program

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

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

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
```

Method 2: Fixed Point Iteration

Goal: Given a function g(x), find a value x_0 such that $g(x_0) = x_0$. Such x_0 is called a **fixed point**.

In our example, finding a root of f(x) = x − cos(x) is equivalent to finding a fixed point of g(x) = cos(x).

> 📌 Not all functions have fixed points, but many root-finding problems can be reformulated as fixed-point problems.

Algorithm

```
Start with some initial point x₀.
Compute x₁ = g(x₀).
If |x₁ - x₀| < ε, stop and return x₁.
Otherwise, set x₀ = x₁ and repeat.
```

Interface File — 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 the number of iterations is exceeded
 *       or |g(xi) - xi| < epsilon, where xi is x0 after i iterations.
 */
double fixed(double x0, double epsilon, int iterations);

#endif
```

Implementation File — fixed.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 Program

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

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

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 Pointers

In the previous examples, the target function was hard-coded. Ideally, the function itself would be passed as a parameter. C supports this via **function pointers**.

Syntax

```
double (*f)(double)   // pointer to a function taking a double and returning a
double
```

📌 The parentheses around (*f) are essential. Without them, double *f(double) would declare a function that returns a pointer to double — a very different thing.

Interface File — bisection2.h

```
#ifndef BISECTION2_H
#define BISECTION2_H
#include <math.h>

/*
 * Pre:  epsilon > 0, iterations > 0,
 *       f(x) has only one root in [a,b], f(a)*f(b) < 0
 * Post: Returns an approximate root of f using the bisection method.
 *       Stops when iterations exceeded or |b - a| < epsilon.
 */
double bisect2(double a, double b, double epsilon, int iterations,
               double (*f)(double));

#endif
```

Implementation File — bisection2.c

```
#include <assert.h>
#include "bisection2.h"
// No need to include math.h here; 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;
}
```

Main Program

```
#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)
{
    // bisect2 called twice, each time with a different function
    printf("%g\n", bisect2(-10, 10, 0.0001, 50, f1));
    printf("%g\n", bisect2(-10, 10, 0.0001, 50, f2));
    return 0;
}
```

Polynomials

A polynomial is an expression with at least one indeterminate and coefficients lying in some set. In general:

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

For example: $3x^3 + 4x^2 + 9x + 2$. We will primarily use integers or doubles for the coefficients.

> ? Think about different ways we can represent polynomials in memory. Consider the pros and cons of each approach before reading further.

Representation

We represent a polynomial as an array of n+1 coefficients where n is the degree. Coefficients are stored in ascending order of power:

```
// 3x³ + 4x² + 9x + 2
double p[] = {2.0, 9.0, 4.0, 3.0};
//             a₀   a₁   a₂   a₃
```

The function signature for polynomial evaluation:

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

Traditional Method

Evaluate $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ directly:

- Compute $x, x^2, x^3, \ldots, x^n$ — requires $n-1$ multiplications.
- Multiply each power by its coefficient $a_1, a_2, \ldots, a_n$ — another n multiplications.
- Add all terms including a_0 — n additions.
- Total: $2n-1$ multiplications and n additions.

> ? Multiplication is computationally more expensive than addition. Is there a way to reduce the number of multiplications?

Horner's Method

Idea: factor the polynomial to remove redundant multiplications.

```
a₀ + a₁x + a₂x² + … + aₙxⁿ
  = a₀ + x(a₁ + x(a₂ + … + x(aₙ₋₁ + aₙx)…))

Example: 2 + 9x + 4x² + 3x³  =  2 + x(9 + x(4 + 3x))
```

Total operations: **n multiplications + n additions** — compared to $2n-1$ multiplications and n additions for the traditional method.

Implementation

```
#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; 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;
}
```

Extra Practice Problems

1.

Write a function named find_sqrt that uses Newton's Iteration to compute √n.

Parameters: double n, double tolerance
Returns: double (the square root)

Start with $x_0 = 1$. Each subsequent guess is:
x_(k+1) = (x_k + n / x_k) / 2

Stop and return x when $|x^2 - n| \leq$ tolerance.

Note: Try to solve this before looking at the provided solution.

```
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 of the following program? Trace manually before running.

```
#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. Cosine via Taylor series:

> Write a program to compute the cosine of x using the Taylor series.
> The user supplies x and a positive integer n. Include all terms up through the term involving x^n.
>
> $\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots$

Solution to problem 1:

```c
#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);
}
```