Approximate computations

Jordi Cortadella
Department of Computer Science

Living with floating-point numbers

- Standard normalized representation (sign + fraction + exponent):

\[ 0.15625_{10} = 0.001012 = 1.01 \times 2^{-3} \]

- Ranges of values:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision (float)</td>
<td>32</td>
<td>±1.18 × 10^{-38} to ±3.4 × 10^{-38}</td>
</tr>
<tr>
<td>double precision (double)</td>
<td>64</td>
<td>±2.23 × 10^{-308} to ±1.80 × 10^{308}</td>
</tr>
</tbody>
</table>

Representations for: \(-\infty, +\infty, +0, -0, NaN\) (not a number)

- Be careful when operating with real numbers:

```cpp
double x, y;
cin >> x >> y; // 1.1 3.1
cout.precision(20);
cout << x + y << endl; // 4.2000000000000001776
```

Single and double-precision FP numbers

float:

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
</table>

= 0.15625

double:

| sign | exponent (11 bits) | fraction (52 bits) |

Comparing floating-point numbers

- Comparisons:

\[ a = b + c; \]

\[ \text{if } (a - b == c) \ldots \] // may be false

- Allow certain tolerance for equality comparisons:

\[ \text{if } (\text{abs}(\text{expr1} - \text{expr2}) < 0.000001) \ldots \] // Ok!

Not every number can be represented.
Example: 0.15 is stored as 0.149999999999999994448884876874

Comparisons:

\[
(-1)^{\text{sign}} \left( 1 - \sum_{i=1}^{52} b_{52-i}2^{-i} \right) \times 2^{e-1023}
\]

or

\[
(-1)^{\text{sign}} \left( 1.0 \right) \times 2^{e-1023}
\]
Root of a continuous function

Bolzano’s theorem:
Let $f$ be a real-valued continuous function. Let $a$ and $b$ be two values such that $a < b$ and $f(a) \cdot f(b) < 0$. Then, there is a value $c \in [a, b]$ such that $f(c) = 0$.

Strategy: narrow the interval $[a, b]$ by half, checking whether the value of $f$ in the middle of the interval is positive or negative. Iterate until the width of the interval is smaller $\varepsilon$.

Design a function that finds a root of a continuous function $f$ in the interval $[a, b]$ assuming the conditions of Bolzano’s theorem are fulfilled. Given a precision ($\varepsilon$), the function must return a value $c$ such that the root of $f$ is in the interval $[c, c+\varepsilon]$.

// Pre: $f$ is continuous, $a < b$ and $f(a) \cdot f(b) < 0$. // Returns $c \in [a, b]$ such that a root exists in the // interval $[c, c+\varepsilon]$.

// Invariant: a root of $f$ exists in the interval $[a, b]$
double root(double a, double b, double epsilon) {
    while (b - a > epsilon) {
        double c = (a + b)/2;
        if (f(a)*f(c) <= 0) b = c;
        else a = c;
    }
    return a;
}

// A recursive version
double root(double a, double b, double epsilon) {
    if (b - a <= epsilon) return a;
    double c = (a + b)/2;
    if (f(a)*f(c) <= 0) return root(a,c,epsilon);
    else return root(c,b,epsilon);
}

The Newton-Raphson method

A method for finding successively approximations to the roots of a real-valued function. The function must be differentiable.

\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \]
The Newton-Raphson method

Square root (using Newton-Raphson)

• Calculate \( x = \sqrt{a} \)

• Find the zero of the following function:
  \[ f(x) = x^2 - a \]
  where \( f'(x) = 2x \)

• Recurrence:
  \[ x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2} \left( x_i + \frac{a}{x_i} \right) \]

Square root (using Newton-Raphson)

// Pre: a >= 0
// Returns x such that \(|x^2-a| < \varepsilon\)

double square_root(double a) {
    double x = 1.0; // Makes an initial guess
    // Iterates using the Newton-Raphson recurrence
    while (abs(x*x - a) >= epsilon) x = 0.5*(x + a/x);
    return x;
}
Approximating definite integrals

• There are various methods to approximate a definite integral:

\[ \int_{a}^{b} f(x) \, dx. \]

• The trapezoidal method approximates the area with a trapezoid:

\[ \int_{a}^{b} f(x) \, dx \approx (b - a) \left( \frac{f(a) + f(b)}{2} \right) \]

The approximation is better if several intervals are used:

```
// Pre: b >= a, n > 0
// Returns an approximation of the definite integral of f
// between a and b using n intervals.
double integral(double a, double b, int n) {
    double h = (b - a)/n;
    double s = 0;
    for (int i = 1; i < n; ++i) s = s + f(a + i*h);
    return (f(a) + f(b) + 2*s)*h/2;
}
```
Monte Carlo methods

• Algorithms that use repeated generation of random numbers to perform numerical computations.

• The methods often rely on the existence of an algorithm that generates random numbers uniformly distributed over an interval.

• In C++ we can use `rand()`, that generates numbers in the interval `[0, RAND_MAX)`

Approximating \( \pi \)

// Pre: n is the number of generated points
// Returns an approximation of \( \pi \) using n random points

double approx_pi(int n) {
    int nin = 0;
    double randmax = double(RAND_MAX);
    for (int i = 0; i < n; ++i) {
        double x = rand()/randmax;
        double y = rand()/randmax;
        if (x*x + y*y < 1.0) nin = nin + 1;
    }
    return 4.0*nin/n;
}

Approximating \( \pi \)

<table>
<thead>
<tr>
<th>n</th>
<th>( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.200000</td>
</tr>
<tr>
<td>100</td>
<td>3.120000</td>
</tr>
<tr>
<td>1,000</td>
<td>3.132000</td>
</tr>
<tr>
<td>10,000</td>
<td>3.171200</td>
</tr>
<tr>
<td>100,000</td>
<td>3.141520</td>
</tr>
<tr>
<td>1,000,000</td>
<td>3.141664</td>
</tr>
<tr>
<td>10,000,000</td>
<td>3.141130</td>
</tr>
<tr>
<td>100,000,000</td>
<td>3.141692</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>3.141604</td>
</tr>
</tbody>
</table>
Generating random numbers in an interval

Assume \( \text{rand()} \) generates a random natural number \( r \) in the interval \([0, R)\).

<table>
<thead>
<tr>
<th>Domain</th>
<th>Interval</th>
<th>Random number</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>[0,1)</td>
<td>( r/R )</td>
</tr>
<tr>
<td>R</td>
<td>[0,a)</td>
<td>( ar/R )</td>
</tr>
<tr>
<td>R</td>
<td>[a,b)</td>
<td>( a + (b - a)r/R )</td>
</tr>
<tr>
<td>Z</td>
<td>[0,a)</td>
<td>( r \mod a )</td>
</tr>
<tr>
<td>Z</td>
<td>[a,b)</td>
<td>( a + r \mod (b - a) )</td>
</tr>
</tbody>
</table>

**Note:** Be careful with integer divisions when delivering real numbers (enforce real division).

---

Monte Carlo applications: examples

- Determine the approximate number of lattice points in a sphere of radius \( r \) centered in the origin.

- Determine the volume of a 3D region \( R \) defined as follows: A point \( P = (x, y, z) \) is in \( R \) if and only if \( x^2 + y^2 + 2z^2 \leq 100 \) and \( 3x^2 + y^2 + z^2 \leq 150 \).

- And many other application domains:
  - Mathematics, Computational Physics, Finances, Simulation, Artificial Intelligence, Games, Computer Graphics, etc.

---

Example: intersection of two bodies

- Determine the volume of a 3D region \( R \) defined as follows: A point \( P = (x, y, z) \) is in \( R \) if and only if \( x^2 + y^2 + 2z^2 \leq 100 \) and \( 3x^2 + y^2 + z^2 \leq 150 \).

- The intersection of the two bodies is inside a rectangular cuboid \( C \), with center in the origin, such that:
  - \( x^2 \leq 50 \)
  - \( y^2 \leq 100 \)
  - \( z^2 \leq 50 \)

- The volume of the cuboid is:

\[
\text{Volume}(C) = (2 \cdot \sqrt{50}) \cdot (2 \cdot 10) \cdot (2 \cdot \sqrt{50}) = 4000
\]

---

Example: intersection of two bodies

// Returns an estimation of the volume of the intersection of two bodies with n random samples.

double volume_intersection(int n) {
    int min = 0;
    double s50 = sqrt(50)/RAND_MAX; // scaling for numbers in [0,sqrt(50))
    double s10 = 10.0/RAND_MAX; // scaling for numbers in [0,10)
    // Generate n random samples
    for (int i = 0; i < n; ++i) {
        // Generate a random point inside the cuboid
        double x = s50*rand();
        double y = s10*rand();
        double z = s50*rand();

        // Check whether the point is inside the intersection
        if (x*x + y*y + 2*z*z <= 100 &&
            3*x*x + y*y + z*z <= 150) ++min;
    }
    return 4000.0*min/n;
}
Example: intersection of two bodies

Summary

• Approximate computations is a resort when no exact solutions can be found numerically.

• Intervals of tolerance are often used to define the level of accuracy of the computation.

• Random sampling methods can be used to statistically estimate the result of some complex problems.