Introduction to Programming (in C++)

Numerical methods I

Jordi Cortadella
Dept. of Computer Science, UPC
Living with floating-point numbers

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

\[ 0.15625_{10} = 0.00101_{2} = 1.01 \times 2^{-3} \]

• Ranges of values:

<table>
<thead>
<tr>
<th>single precision (float)</th>
<th>double precision (double)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>±1.18 \times 10^{-38} to ±3.4 \times 10^{38}</td>
<td>±2.23 \times 10^{-308} to ±1.80 \times 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
```
Comparing floating-point numbers

• Comparisons:

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

• Allow certain tolerance for equality comparisons:

\[
if (\text{expr1} == \text{expr2}) \ldots \quad // \text{Wrong} !
\]
\[
if (\text{abs}(\text{expr1} - \text{expr2}) < 0.000001) \ldots \quad // \text{Ok} !
\]
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 $[0, \text{RAND\_MAX}]$.
Approximating $\pi$

- Let us pick a random point within the unit square.
- **Q:** What is the probability for the point to be inside the circle?
- **A:** The probability is $\pi/4$

**Algorithm:**
- Generate $n$ random points in the unit square
- Count the number of points inside the circle ($n_{\text{in}}$)
- Approximate $\pi/4 \approx n_{\text{in}}/n$
#include <cstdlib>

// 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>3.141692</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>
The Newton-Raphson method

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

\[ \tan \alpha = f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}} \]

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

source: http://en.wikipedia.org/wiki/Newton’s_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:
  \[
  x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{1}{2} \left( x_i + \frac{a}{x_i} \right)
  \]
// Pre: a ≥ 0
// Returns x such that |x^2 - a| < ε

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;
}
Square root (using Newton-Raphson)

• Example: `square_root(1024.0)`

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000000000000000000</td>
</tr>
<tr>
<td>512.5000000000000000000</td>
</tr>
<tr>
<td>257.24902439024390332634</td>
</tr>
<tr>
<td>130.61480157022683101786</td>
</tr>
<tr>
<td>69.227324054488946103447</td>
</tr>
<tr>
<td>42.009585631008270922848</td>
</tr>
<tr>
<td>33.192487416854376647279</td>
</tr>
<tr>
<td>32.021420905000240964000</td>
</tr>
<tr>
<td>32.000007164815897908738</td>
</tr>
<tr>
<td>32.000000000000802913291</td>
</tr>
</tbody>
</table>
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:

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

- The approximation is better if several intervals are used:
Approximating definite integrals

\[ S = \sum_{i=0}^{n-1} h \cdot \frac{f(a_i) + f(a_{i+1})}{2} \]

\[ = \frac{h}{2} \left( f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(a_i) \right) \]
Approximating definite integrals

// 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;
}
A polygon can be represented by a sequence of vertices.

Two consecutive vertices represent an edge of the polygon.

The last edge is represented by the first and last vertices of the sequence.

Vertices:  \( (1,3) \) \( (4,1) \) \( (7,3) \) \( (5,4) \) \( (6,7) \) \( (2,6) \)

Edges:  \( (1,3) \)-\( (4,1) \)-\( (7,3) \)-\( (5,4) \)-\( (6,7) \)-\( (2,6) \)-\( (1,3) \)
• Is a point inside a polygon?
• Use the crossing number algorithm:
  ▪ Draw a ray from the point
  ▪ Count the number of crossing edges:
    ➢ even → outside, odd → inside.
// A data structure to represent a point
struct Point {
    double x;
    double y;
};

// A data structure to represent a polygon
// (an ordered set of vertices)
typedef vector<Point> Polygon;
Point in polygon

- Use always the horizontal ray increasing $x$ ($y$ is constant)
- Assume that the probability of “touching” a vertex is zero

\[
\frac{y_2 - y_1}{x_2 - x_1} = \frac{y - y_1}{x_c - x_1} \\
\downarrow \\
x_c = x_1 + \frac{y - y_1}{y_2 - y_1} (x_2 - x_1)
\]

- The ray crosses the segment if:
  - $y$ is between $y_1$ and $y_2$ and
  - $x_c > x$
// Returns true if point q is inside polygon P,
// and false otherwise.

bool in_polygon(const Polygon& P, const Point& q) {
    int nvert = P.size();
    int src = nvert - 1;
    int ncross = 0;

    // Visit all edges of the polygon
    for (int dst = 0; dst < nvert; ++dst) {
        if (cross(P[src], P[dst], q) ++ncross;
            src = dst;
    }

    return ncross%2 == 1;
}
// Returns true if the horizontal ray generated from q by increasing x crosses the segment defined by p1 and p2, and false otherwise.

bool cross(const Point& p1, const Point& p2, const Point& q) {

    // Check whether q.y is between p1.y and p2.y
    if ((p1.y > q.y) == (p2.y > q.y)) return false;

    // Calculate the x coordinate of the crossing point
    double xc = p1.x + (q.y - p1.y)*(p2.x - p1.x)/(p2.y - p1.y);
    return xc > q.x;
}
Cycles in permutations

- Let $P$ be a vector of $n$ elements containing a permutation of the numbers 0...$n-1$.
- The permutation contains cycles and all elements are in some cycle.

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[i]$</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Cycles:
- $(0 \ 6 \ 9 \ 1 \ 4)$
- $(3 \ 8 \ 5 \ 7)$
- Design a program that writes all cycles of a permutation.
Cycles in permutations

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P[i]$</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$visited[i]$</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

- Use an auxiliary vector (visited) to indicate the elements already written.
- After writing one permutation, the index returns to the first element.
- After writing one permutation, find the next non-visited element.
Cycles in permutations

\[
\begin{align*}
// \textbf{Pre:} & \quad P \text{ is a vector with a permutation of } 0..n-1 \\
// \textbf{Post:} & \quad \text{The cycles of the permutation have been printed in cout}
\end{align*}
\]

```cpp
void print_cycles(const vector<int>& P) {
    int n = P.size();
    vector<bool> visited(n, false);
    int i = 0;
    while (i < n) {
        // All the cycles containing 0..i-1 have been written
        bool cycle = false;
        while (not visited[i]) {
            if (not cycle) cout << '(';  
            else cout << ');' ; // Not the first element
            cout << i;
            cycle = true;
            visited[i] = true;
            i = P[i];
        }
        if (cycle) cout << ')' << endl;
        i = i + 1;
    }
}
```

---

Introduction to Programming © Dept. CS, UPC 26
Taylor and McLaurin series

• Many functions can be approximated by using Taylor or McLaurin series, e.g.:

\[ f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n = f(0) + f'(0)x + \frac{f''(0)}{2!} x^2 + \frac{f^{(3)}(0)}{3!} x^3 + \cdots + \frac{f^{(n)}(0)}{n!} x^n + \cdots \]

• Example: \( \sin x \)

\[ \sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n + 1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \]
Calculating $\sin x$

- McLaurin series:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n + 1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

- It is a periodic function (period is $2\pi$)

- Convergence improves as $x$ gets closer to zero
Calculating $\sin x$

- Reducing the computation to the $(-2\pi, 2\pi)$ interval:

\[ k = \left\lfloor \frac{x}{2\pi} \right\rfloor, \quad \sin x = \sin(x - 2k\pi). \]

- Incremental computation of terms:

\[ t_i = \frac{(-1)^i x^{2i+1}}{(2i + 1)!}, \quad t_{i+1} = \frac{(-1)^{i+1} x^{2i+3}}{(2i + 3)!} = -t_i \cdot \frac{x^2}{(2i + 2)(2i + 3)} \]
#include <cmath>

// Returns an approximation of sin x.
double sin_approx(double x) {
    int k = int(x/(2*M_PI));
    x = x - 2*k*M_PI; // reduce to the (-2π,2π) interval
    double term = x;
    double x2 = x*x;
    int d = 1;
    double sum = term;

    while (abs(term) >= 1e-8) {
        term = -term*x2/((d+1)*(d+2));
        sum = sum + term;
        d = d + 2;
    }

    return sum;
}
Lattice paths

We have an \( n \times m \) grid.

How many different routes are there from the bottom left corner to the upper right corner only using right and up moves?
Lattice paths

Some properties:

- $\text{paths}(n, 0) = \text{paths}(0, m) = 1$
- $\text{paths}(n, m) = \text{paths}(m, n)$
- If $n > 0$ and $m > 0$:
  
  $$\text{paths}(n, m) = \text{paths}(n-1, m) + \text{paths}(n, m-1)$$
// **Pre:** n and m are the dimensions of a grid
//       (n ≥ 0 and m ≥ 0).
// **Returns** the number of lattice paths in the grid.

```c
int paths(int n, int m) {
    if (n == 0 || m == 0) return 1;
    return paths(n - 1, m) + paths(n, m - 1);
}
```
How large is the tree (cost of the computation)?
Observation: many computations are repeated
### Lattice paths

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>20</td>
<td>35</td>
<td>56</td>
<td>84</td>
<td>120</td>
<td>165</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>35</td>
<td>70</td>
<td>126</td>
<td>210</td>
<td>330</td>
<td>495</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>6</td>
<td>21</td>
<td>56</td>
<td>126</td>
<td>252</td>
<td>462</td>
<td>792</td>
<td>1287</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>7</td>
<td>28</td>
<td>84</td>
<td>210</td>
<td>462</td>
<td>924</td>
<td>1716</td>
<td>3003</td>
</tr>
</tbody>
</table>

\[
M[i][0] = M[0][i] = 1
\]

\[
M[i][j] = M[i - 1][j] + M[i][j - 1], \quad \text{for } i > 0, j > 0
\]
Lattice paths

// **Pre:** n and m are the dimensions of a grid
//      \((n \geq 0 \text{ and } m \geq 0)\).
// Returns the number of lattice paths in the grid.

int paths(int n, int m) {
    vector<vector<int>> M(n + 1, vector<int>(m + 1));
    // Initialize row 0
    for (int j = 0; j <= m; ++j) M[0][j] = 1;

    // Fill the matrix from row 1
    for (int i = 1; i <= n; ++i) {
        M[i][0] = 1;
        for (int j = 1; j <= m; ++j) {
            M[i][j] = M[i - 1][j] + M[i][j - 1];
        }
    }
    return M[n][m];
}
Lattice paths

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

\[ M[i][j] = \binom{i+j}{i} = \binom{i+j}{j} \]
In a path with $n+m$ segments, select $n$ segments to move right (or $m$ segments to move up)

- Subsets of $n$ elements out of $n+m$
Lattice paths

Calculating \( \binom{n}{k} = \frac{n!}{k!(n-k)!} \)

– Naïve method: \(2n\) multiplications and 1 division (potential overflow problems with \(n!\))

– Recursion:

\[
\binom{n}{0} = \binom{n}{n} = 1
\]

\[
\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1} = \frac{n-k+1}{k} \binom{n}{k-1}
\]

\[
= \frac{n}{n-k} \binom{n-1}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]
Lattice paths

// Pre: n and m are the dimensions of a grid
//      (n ≥ 0 and m ≥ 0).
// Returns the number of lattice paths in the grid.

int paths(int n, int m) {
    return combinations(n + m, n);
}

// Pre: n ≥ k ≥ 0
// Returns the number of k-combinations of a set of
// n elements.

int combinations(int n, int k) {
    if (k == 0) return 1;
    return n*combinations(n - 1, k - 1)/k;
}
Lattice paths

Computational cost:

– Recursive version: $O \left( \binom{n + m}{m} \right)$

– Matrix version: $O(n \cdot m)$

– Combinations: $O(m)$
Lattice paths

• How about counting paths in a 3D grid?
• And in a k-D grid?