

PHY 410-505 Computational Physics I

Chapter 1: A First Numerical Problem

Lecture 3

Friday August 29, 2008

Lecture Outline

Errors, Accuracy, and Stability	3
Range Errors	4
Accuracy	6
Standard for Floating Point Arithmetic	6
Roundoff Errors	9
Local and Global Errors	14
Instabilities	15

Errors, Accuracy, and Stability

See discussion in Numerical Recipes Section 1.3

Truncation Errors: also called algorithm errors – e.g., Euler's algorithm

$$\begin{aligned}x(t_n + \Delta t) &= x(t_n) + \left. \frac{dx}{dt} \right|_{t_n} \Delta t + \frac{1}{2} \left. \frac{d^2x}{dt^2} \right|_{t_n} \Delta t^2 + \dots \\ &= x_n + \Delta t v_n + \mathcal{O}(\Delta t^2)\end{aligned}$$

Range Errors: data types `int`, `double` have limits – computer cannot represent value

Accuracy: finite precision (significant figures) – causes roundoff errors – affect arithmetic operations like subtraction – can accumulate if computation has many steps

Instabilities: algorithm generates wrong solution due to truncation or roundoff errors

Range Errors

C++: the standard only specifies minimum requirements for integer and real types – actual values depend on operating system and compiler – the code `range.cpp` queries some of these limits

`range.cpp`

```
#include <iostream>
#include <limits>
using namespace std;

int main() {
    cout << "Biggest int      = " << numeric_limits<int>::max() << endl
         << "Smallest int     = " << numeric_limits<int>::min() << endl
         << "Biggest long     = " << numeric_limits<long>::max() << endl
         << "Smallest long    = " << numeric_limits<long>::min() << endl
         << "Biggest float    = " << numeric_limits<float>::max() << endl
         << "Smallest float   = " << numeric_limits<float>::min() << endl
         << "Biggest double   = " << numeric_limits<double>::max() << endl
         << "Smallest double  = " << numeric_limits<double>::min() << endl;
}
```

If your program generates values outside these ranges it will continue to run with incorrect results!

Mathematica: can do arbitrary precision arithmetic!

remainders right to left as binary digits:

$$0 + 1 \leftarrow 1 + 1 \leftarrow 3 + 1 \leftarrow 7 + 1 \leftarrow 15 + 1 \leftarrow 31 + 1 \leftarrow 63 + 1 \leftarrow 127 + 1 \leftarrow 255 + 1 \leftarrow 511 + 0 \leftarrow 1022 = 1111111110$$

- E.g, the rational fraction $7/5$ which = 1.4 exactly in the decimal system, it is represented by:

s	e	m
0	0111111111	011001100110011001100110011001100110011001100110

0.4000... in decimal = $2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + \dots$ requires an infinite number of binary digits to specify, just like $1/3 = 0.3333\dots$ in decimal

- To convert decimal 0.4 to binary repeatedly multiply by 2 and record the integer parts left to right: $0.4 \rightarrow 0 + 0.8 \rightarrow 1 + 0.6 \rightarrow 1 + 0.2 \rightarrow 0 + 0.4 \rightarrow 0 + 0.8 \rightarrow 1 + 0.6 \rightarrow 1 + 0.2 \rightarrow + \dots = 0.011001100110\dots$

The code `doublebits.cpp`, analyzes the bits in a double, and also reveals the “Endianness” of your computer:

`doublebits.cpp`

```
#include <iostream>
```

```
#include <iomanip>
#include <limits>
using namespace std;

void printBitsInByte (unsigned char *byte, ostream& os) {
    // print bits from left to right
    for (int i = 7; i >= 0; --i)
        os << (int) ((*byte >> i) & (unsigned char) 1);
    os << ' '; // followed by a space
}

int main() {
    cout << "\nRoundoff in representing rational fraction as double";
    cout << "\n-----";

    int n, d;
    cout << "\nEnter integer numerator: " << flush;
    cin >> n;
    cout << "Enter integer denominator: " << flush;
    cin >> d;
    double x = n / (double) d;

    // get address of x and pretend it points to a byte (8 bits)
    unsigned char *byte = (unsigned char *) &x;
    cout << "\nThe bit representation of " << n << "/" << d << " ";
```

```

cout << "assuming a Big Endian computer is:\n ";
for (int i = 0; i < 8; i++)
    printBitsInByte(byte + i, cout);
cout << "\nand, assuming a Little Endian computer is:\n ";
for (int i = 7; i >= 0; i--)
    printBitsInByte(byte + i, cout);

cout.precision(20);
cout << "\n\nand its decimal value = " << x << endl;
}

```

Roundoff Errors

- Errors due to finite precision can accumulate during a calculation – the code `manyround.cpp` illustrates this by comparing x with $\frac{1}{N} \sum_{i=1}^N x$ using single precision `float` and double precision `double`.

`manyround.cpp`

```

#include <iostream>
#include <iomanip>
#include <limits>
using namespace std;

int main () {

```

```

cout << "\nAccumulation of many roundoff errors"
    << "\n-----"
    << "\nAdd x to itself N times, and"
    << "\nthen divide the result by N"
    << "\n\nEnter value of x: " << flush;
double x;
cin >> x;
cout << "Enter number of times N: " << flush;
int N;
cin >> N;
double ans = x;
float y = x, fans = x;
for (int i = 1; i < N; i++) {
    ans += x;
    fans += y;
}
ans /= N;
fans /= N;
cout.precision(16);
cout.setf(ios::left);
cout << "\n          Double Precision          Single Precision  "
    << "\n-----"
    << "\nAnswer    " << setw(22) << ans
    << "    " << setw(22) << fans
    << "\nError      " << setw(22) << (ans - x)

```

```

    << "  " << setw(22) << (fans - y)
    << "\nEpsilon  " << setw(22) << numeric_limits<double>::epsilon()
    << setw(22) << numeric_limits<float>::epsilon() << endl;
}

```

- Roundoff can also cause serious loss of precision in familiar mathematical formulas like

$$ax^2 + bx + c = 0, \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

If b is positive and $ac \ll b^2$, the numerator is the difference of two nearly equal numbers – causes loss of accuracy (significant figures)

- One solution is to check whether $b > 0$ and compute the smaller root using product of roots = c/a
- The code `quadratic.cpp` illustrates this problem.

`quadratic.cpp`

```

#include <iostream>
#include <iomanip>
#include <cmath>

```

```
using namespace std;

int main () {
    cout << "\nRoundoff errors in quadratic formula"
         << "\n-----"
         << "\nSolve quadratic  ax^2 + bx + c = 0"
         << "\nUsing (-b +- sqrt(b^2 - 4ac) / (2a)"
         << "\nCompare with better smaller = c / (a * larger)"
         << "\n\nEnter two roots x1 x2: " << flush;
    double x1, x2;
    cin >> x1 >> x2;

    double a = 1;
    double b = - (x1 + x2);
    double c = x1 * x2;
    double discr = b * b - 4 * a * c;
    cout << "a = " << a << ", b = " << b << " c = " << c
         << "\ndiscriminant b^2 - 4ac = " << discr << endl;

    double larger = (-b + sqrt(discr)) / (2 * a);
    double smaller = (-b - sqrt(discr)) / (2 * a);
    if (b > 0) {
        double swap = larger;
        larger = smaller;
        smaller = swap;
    }
}
```

```

}
double better = c / (a * larger);
if (abs(x1) < abs(x2)) {
    double swap = x1;
    x1 = x2;
    x2 = swap;
}

cout.precision(16);
cout.setf(ios::left);
cout << "-----\n"
    << "          Value of root          Fractional error          \n"
    << "-----\n";
cout << "Larger |root|  " << setw(22) << larger << "  "
    << abs((larger - x1) / x1) << '\n'
    << "Smaller |root|  " << setw(22) << smaller << "  "
    << abs((smaller - x2) / x2) << '\n'
    << "Better smaller  " << setw(22) << better << "  "
    << abs((better - x2) / x2) << '\n'
    << "-----\n";
}

```

Local and Global Errors

- Numerical algorithms usually involve many steps
- Each step has
 1. an algorithm (truncation) error
 2. roundoff errors due to arithmetic operations
- The sum of the errors in each step is the *local error*
- Errors in each step will *accumulate* to produce a *global error* for the calculation. If the calculation has N steps
 - local errors with random signs usually results in a global error $\sim \sqrt{N} \times$ average local error
 - local errors with mostly the same sign can results in a larger global error $\sim N \times$ average local error

Instabilities

- An algorithm might have an instability – roundoff leads to the wrong solution
- Numerical Recipes gives a simple example involving the Golden Mean

$$\phi \equiv \frac{\sqrt{5} - 1}{2} \approx 0.61803398 \dots$$

- Can use

$$\phi^{n+1} = \phi^{n-1} - \phi^n$$

to compute ϕ^2, ϕ^3, \dots using subtraction instead of multiplication (subtraction is faster!)

- The recursion has another solution

$$-\phi \equiv \frac{\sqrt{5} + 1}{2} \approx 0.61803398 \dots$$

- The code `golden.cpp` compares the recursion result with the desired result.

```
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    double phi = (sqrt(5.0) - 1) / 2;
    double phi_nMinus1 = 1, phi_n = phi;

    cout << "n\tRecursion\tMultiplication\tPercent Error" << endl;
    for (int n = 1; n < 45; n++) {
        double phi_mult = pow(phi, n);
        double percentError = abs((phi_n - phi_mult) / phi_mult * 100);
        cout << n << '\t' << phi_n << '\t' << phi_mult << '\t'
            << percentError << '\n';
        double phi_nPlus1 = phi_nMinus1 - phi_n;
        phi_nMinus1 = phi_nMinus1 = phi_n;
        phi_n = phi_nPlus1;
    }
}
```