Introduction to Scientific Computing

Robert P. Goddard

Applied Physics Laboratory
University of Washington
Seattle, WA

Northwest C++ User's Group
16 November 2011

Outline

Richard W. Hamming's Five Main Ideas

The purpose of computing is insight, not numbers.

R. W. Hamming, *Numerical Methods for Scientists and Engineers, Second Edition*, McGraw-Hill, New York etc. (1973), Chapter 1: "An Essay on Numerical Methods"

- 0. *Numbers*: Counting, fixed-point, floating-point (Hamming Chapter 2)
- 1. Computing is intimately bound up with both the source of the problem and the use that is going to be made of the answers – it is not a step to be taken in isolation from reality.
- 2. It is necessary to study families and to relate one family to another when possible, and to avoid isolated formulas and isolated algorithms.

Hamming's Five Main Ideas

continued

- 3. *Roundoff error*: The greatest loss of significance in the numbers occurs when two numbers of about the same size are subtracted so that most of the leading digits cancel out.
- 4. *Truncation error*: Many of the processes of mathematics, such as differentiation and integration, imply the use of a limit which is an infinite process. The machine can only do a finite number of operations in a finite length of time.
- 5. *Feedback*: Numbers at one stage are fed back into the computer to be processed again and again. Feedback leads to the idea of *stability* of the feedback loop – will a small error grow or decay through the successive iterations?

Outline

Scientific Computing Is...

Computing that models part of the physical world.

- Typical Inputs:
  - Detailed measurements of the environment or system
  - Random potential values of imperfectly known quantities
  - Parameter values to be determined by fitting measurements
- Typical Outputs:
  - Visual representations of data: Images, graphs, etc.
  - Sound, radio waves, or other signals
  - Predictions of success or failure of a system
  - Actions to control other devices or processes
  - Parameter values determined by fitting measurements
- Typical Algorithms:
  - Solving sets of differential equations
  - Integrating functions over multi-dimensional domains
  - Optimization, e.g. data fitting
  - Linear algebra (eigenvalues, solvers, etc.)

Scientific Computing Requirements

Speed and Realism

- Typical Speed Requirements:
  - Real Time (hard or soft)
  - Much faster than real time (e.g. for Monte Carlo studies)
  - Fast enough to fit project cost and schedule
- Accurate Enough to Accomplish the Mission:
  - Personnel training
  - Performance prediction
  - Advance of scientific knowledge
  - Control devices or processes
  - Life or death
- What the Computer Does, Mostly:
  - Read floating point numbers from memory
  - Multiply and divide floating point numbers
  - Add and subtract floating point numbers
  - Write floating point numbers to memory

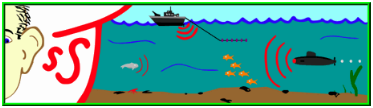
Scientific Computing Requirements

Speed and Realism

- Typical Speed Requirements:
  - Real Time (hard or soft)
  - Much faster than real time (e.g. for Monte Carlo studies)
  - Fast enough to fit project cost and schedule
- Accurate Enough to Accomplish the Mission:
  - Personnel training
  - Performance prediction
  - Advance of scientific knowledge
  - Control devices or processes
  - Life or death
- What the Computer Does, Mostly:
  - Read floating point numbers from memory
  - Multiply and divide floating point numbers
  - Add and subtract floating point numbers
  - Write floating point numbers to memory

Outline

My Example: Sonar Simulation Toolset (SST)



- *Inputs*: Detailed descriptions of sound speed, surface, bottom, bathymetry, multi-channel listening system, sound sources & reflectors (man-made & natural) with trajectories, etc.
- *Output*: Digital representation of sound, suitable for input to signal processing systems including human ears and brains.
- *Applications*: Testing sonar and communication systems, testing ideas for systems that don't exist yet, training sonar operators, understanding observations, predicting system performance in environment where we can't go, etc.
- *Processing*: Acoustical models, geometry, random numbers, signal filters & delays, etc. Front end is a parser.

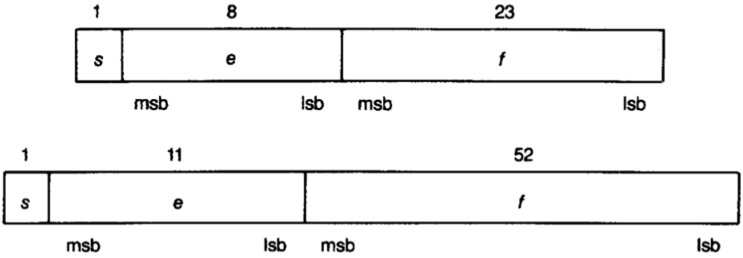
Outline

Format Parameters
IEEE 754-2008

Name	Common name	Base	Digits	E min	E max	Notes	Decimal digits	Decimal E max
binary16	Half precision	2	10+1	−14	+15	storage, not basic	3.31	4.51
binary32	Single precision	2	23+1	−126	+127		7.22	38.23
binary64	Double precision	2	52+1	−1022	+1023		15.95	307.95
binary128	Quadruple precision	2	112+1	−16382	+16383		34.02	4931.77
decimal32		10	7	−95	+96	storage, not basic	7	96
decimal64		10	16	−383	+384		16	384
decimal128		10	34	−6143	+6144		34	6144

Figure from Wikipedia

Format Layout
IEEE 754



Exponent Bias: $e = E + E_{\max}$: $1 \leq e \leq 2^w - 2$
Figures from IEEE 754-1985

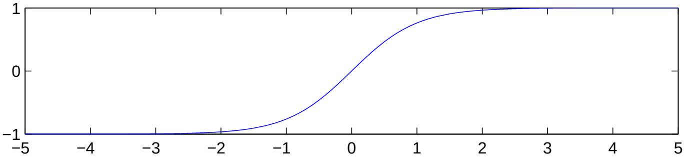
Values
IEEE 754

1. *Not A Number*: If $e = 2^w - 1$ and $f \neq 0$, then v is NaN regardless of s. There are $2^b - 2$ different NaNs: signaling/quiet, "retrospective diagnostic information."
2. *Signed Infinity*:If $e = 2^w - 1$ and $f = 0$, then $v = (-1)^s\infty$
3. *Normalized Numbers*: If $0 < e < 2^w - 1$, then $v = (-1)^s 2^{e-\text{bias}}(1 \cdot f)$
4. *Denormalized Numbers*: If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{1-\text{bias}}(0 \cdot f)$
5. *Signed Zero*: If $e = 0$ and $f = 0$, then $v = (-1)^s 0$

Quiz Question 1

Hyperbolic Tangent

$$\tanh(x) \equiv \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Give 3 reasons not to use that formula to compute tanh.

Answer 1: Overflow

Hyperbolic Tangent

$$\tanh(x) \equiv \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$x = 709 \Rightarrow e^x \approx 8.2 \times 10^{307} \qquad \Rightarrow \tanh(x) = 1$$
$$x = 710 \Rightarrow e^x = \text{Inf} \qquad \Rightarrow \tanh(x) = \text{NaN}$$

If x is a double. Possible solution: Underflow is usually better then overflow:

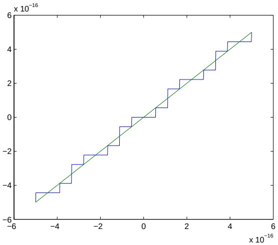
$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

But what about $x < 0$? You need an "if" or an "abs" and a "sign". Or, test for range of $|x|$ and just set the answer to ± 1 if it's big.

Answer 2: Round-off and Subtraction
Hyperbolic Tangent

$$\tanh(x) \equiv \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Look closely around $0 \pm 5 \times 10^{-16}$ (double precision):



Solution: Use a power series for small x. That's yet another "if".

Answer 3: It's in the library!

Hyperbolic Tangent

$\tanh(x)$ is in the library: FORTRAN 77 Intrinsics, C 1990, C++ 1998, etc.

- Small values: Green line.
- $\tanh(10000.0) = 1.0$

Don't re-invent the wheel.

Quiz Question 2

Inverses

```
bool q2( double x, double y)
{
    return x == y && 1.0/x != 1.0/y;
}
```

Can this function ever return *true*? If so, what are x and y?

Quiz Question 2

Inverses

```
bool q2( double x, double y)
{
    return x == y && 1.0/x != 1.0/y;
}
```

Can this function ever return *true*? If so, what are x and y?

```
    double x = 0.0; double y = -0.0;
    bool comp2 = q2( x, y );
    std::cout << "q2( " << x << ", " << y << " ) = " << comp2
        << "    1.0/x = " << 1.0/x
        << "    1.0/y = " << 1.0/y << std::endl;
```

```
q2( 0, -0 ) = 1   1.0/x = inf   1.0/y = -inf
```

Quiz Question 3
Comparisons

```
bool q3( double x, double y)
{  return x == y || x < y || x > y; }
```

Can this function ever return *false*? If so, what are x and y?

Quiz Question 3

Comparisons

```
bool q3( double x, double y)
{  return x == y || x < y || x > y; }
```

Can this function ever return *false*? If so, what are x and y?

```
    double x = 0.0; double y = -0.0;
    double z = x/y;
    bool comp3 = q3( z, z );
    std::cout << "q3( " << z << ", " << z << " ) = "
        << comp3 << std::endl;
```

```
q3( nan, nan ) = 0
```

Every NaN is unordered. No matter what ordering operator you use, the answer is always *false* if either operand is a NaN.

Outline

Example: Second-Order ODE with Initial Conditions
Simple Algorithm

General Problem:

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = \mathrm{f}(x, y)$$

Test Problem (answer is $A \cos x + B \sin x$):

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = -y$$

Simple Solution Approach

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = \lim_{h \to 0} \frac{y(x-h) - 2y(x) + y(x+h)}{h^2}$$

$$\begin{aligned} y(x+h) &\approx 2y(x) - y(x-h) + h^2 \frac{\mathrm{d}^2 y}{\mathrm{d}x^2}(x) \\ &= 2y(x) - y(x-h) + h^2 \mathrm{f}(x, y) \end{aligned}$$

Example: Second-Order ODE with Initial Conditions

Simple Algorithm: C++ Implementation

```
// Return the second derivative: y'' = -y
// which should compute sin(x) or cos(x)
template<typename T>
T d2sine( T x, T y )
{  return -y; }

// Solver for ODE of form y'' = f(x,y)
// First try: Low order, Roundoff problems
template<typename T>
T solve2a(
    T (*d2)(T,T), // RHS
    T x0, T y0,    // Initial y(x0)
    T y1, T y2,    // Initial y(x0+h) and y(x0+2*h)
    T h,  size_t n     // Step size, Number of steps
)
{
    T x = x0 + h; T yprev = y0;  T y = y1;
    for ( size_t i = 2u; i <= n; ++i ) {
        T ynext = 2.0*y - yprev + h*h*d2(x,y);
        yprev = y; y = ynext; x += h;

    }
    return y;
}
```

Example: Second-Order ODE with Initial Conditions

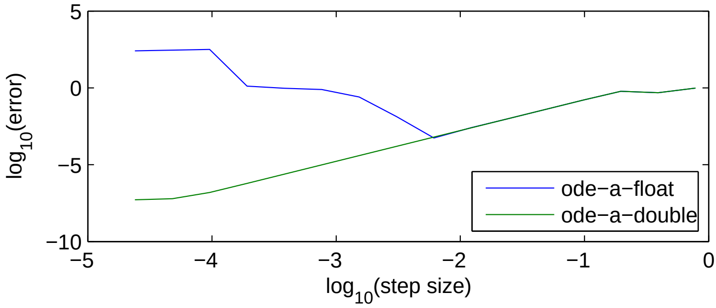
Simple Algorithm: C++ Test Code

```
// Test Script for Second-Order ODE Solver
template<typename T>
void test(
    T (*solver)( T (*)(T,T), T, T, T, T, T, size_t ),
    T (*d2)(T,T)  // RHS
)
{
    const size_t n_cycles = 64;
    const size_t n_trials = 16;
    std::cout << "    h       y" << std::endl;
    T h = std::atan(T(1.0));    // Start at pi/4
    size_t n = 8*n_cycles;    // Number of steps
    T x0 = 0.0f;
    T y0 = 0.0f;
    for ( size_t itrial = 0; itrial < n_trials; ++itrial ) {
        T y1 = std::sin( h );
        T y2 = std::sin( h+h );
        T y = solver( d2, x0, y0, y1, y2, h, n );
        std::cout << h << " " << y << std::endl;
        n *= 2u;
        h /= 2.0f;
    }
}
```

Example: Second-Order ODE with Initial Conditions

Simple Algorithm: Result

```
test( solve2a<float>, d2sine<float> );
test( solve2a<double>, d2sine<double> );
```



Outline

Example: Second-Order ODE with Initial Conditions

Second Algorithm: Better round-off

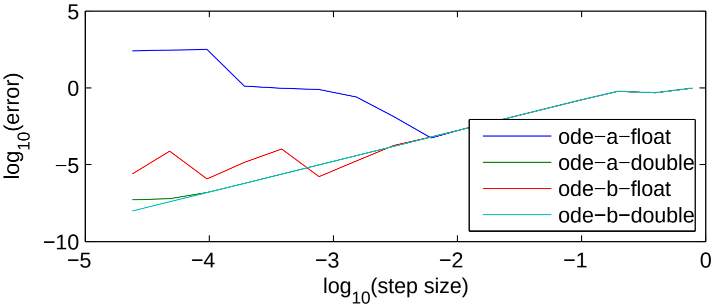
This version avoids excessive round-off.

```
// Solver for ODE of form y'' = f(x,y)
// Second try: Low order, better roundoff
template<typename T>
T solve2b(
         T (*d2)(T,T), // RHS
    T x0, T y0,    // Initial y(x0)
    T y1, T y2,    // Initial y(x0+h) and y(x0+2*h)
    T h,  size_t n     // Step size, Number of steps
 )
{
    T x = x0 + h;
    T y = y1;
    T dy = y1 - y0;
    for ( size_t i = 1; i < n; ++i ) {
        dy += h*h*d2(x,y);
        y += dy;
        x += h;
    }
    return y;
}
```

Example: Second-Order ODE with Initial Conditions

Second Algorithm: Result

```
test( solve2b<float>, d2sine<float> );
test( solve2b<double>, d2sine<double> );
```



Outline

Example: Second-Order ODE with Initial Conditions

Third Algorithm: Higher order

This version has higher order, with error of order h^5, but it's unstable!

```
// Third try: Higher order, unstable: Predictor from Abramowitz & Stegun 25.5.15
template<typename T>
T solve2c(
    T (*d2)(T,T), // RHS
    T x0, T y0,    // Initial y(x0)
    T y1, T y2,    // Initial y(x0+h) and y(x0+2*h)
    T h,  size_t n     // Step size, Number of steps
         )
{
    T x = x0 + h; T y = y2;  T ym1 = y1; T ym2 = y0;
    for ( size_t i = 2; i < n; ++i ) {
        T xm1 = x;
        x = x0 + i*h;
        T yp1 = ym2 + T(3.0)*(y - ym1)
            + h*h*(d2(x,y) - d2(xm1, ym1));
        ym2 = ym1;
        ym1 = y;
        y = yp1;
    }
    return y;
}
```

Example: Second-Order ODE with Initial Conditions

Third Algorithm: Unstable!

```
test( solve2c<float>, d2sine<float> );
test( solve2c<double>, d2sine<double> );
```