计算物理

Lecture 4

傅子文

Outline of Lecture 4

- Discussion on projects, and general approach to numerical methods
- More on floating point – special values & exceptions
  - Special values
  - Exceptions (e.g. Overflow, underflow)
- Problems with floating point calculations
  - Rounding
  - Errors & error propagation
  - Effects in the real world

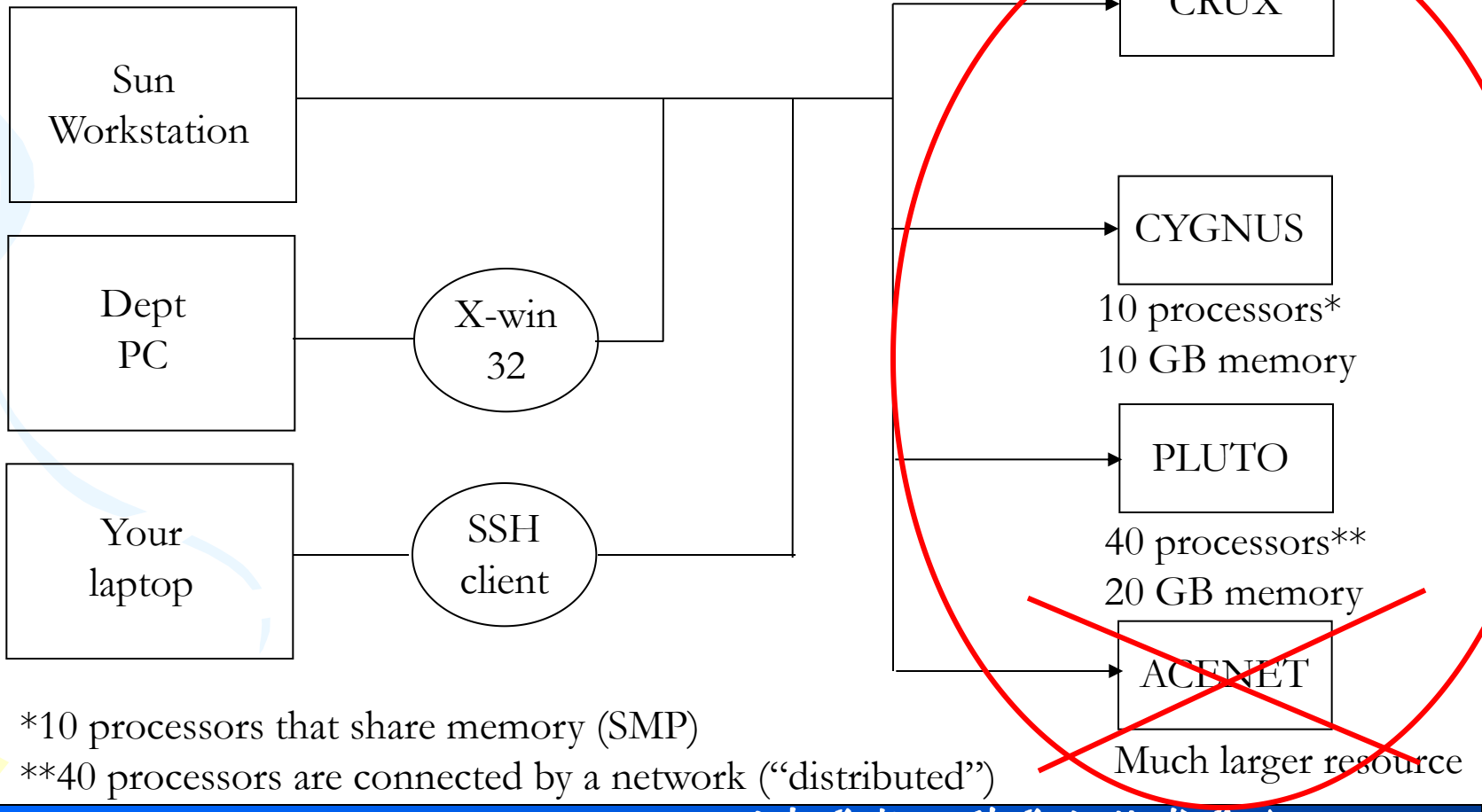
Key steps in any scientific computation

1. **Encapsulate the problem**
   What needs to be solved? What are your assumptions? How general do solutions need to be?
2. **Work analytically as far as you can**
   Stop when you reach transcendental functions, untractable integrations, or differential equations that cannot be solved
3. **Identify what needs to be solved numerically**
   Do you need to find roots? Do you need a numerical integrator? Solve PDEs or ODEs?
4. **Determine a suitable algorithm**
   You may need to derive something new, or alternatively rely on a trusted method
5. **Write the computer program**
   Decide on language, how much functionality is required, can you use existing codes?
6. **Solve**
7. **Interpret your results**
   Can you come up with an analytical approximation? Is the numerical solution sensible? Error checking in numerical methods is very important!

Computational Resources

- Does anyone need a departmental account?

Solaris or Linux



*10 processors that share memory (SMP)

**40 processors are connected by a network ("distributed")

Special values in IEEE 754

- Certain values of the exponent and mantissa are reserved for *special values*
- **Zero**: Mantissa is all 0, exponent is all 0
  - There are two zeros though, depending upon sign bit
- **Infinity**: there are two signed possibilities, both have mantissa all 0 and exponents all 1
  - Result is usually written +INF or –INF
  - Calculation will still attempt to continue with this value

NaN: not a number

- Result for illegal operation
  - i.e. square root of a negative number or division of 0 by 0
- Exponent is maximum value, fraction is non zero (can be anything)
- INF and NaN are not the same
  - Can view INF as a subset of NAN
  - Again, NaN will not necessarily stop the computation from continuing

Operator	Case
+	INF+(-INF)
×	0×INF
/	0/0,INF/INF
√	√x for x<0

Exceptions

- There are 5 different exceptions
  - **Overflow**
    - Exponent of the result is larger than the maximum allowed value
  - **Underflow**
    - Denormalized number results (but depends on precise handling by the machine)
  - **Divide by 0**
    - Infinite result
  - **Invalid operations**
    - Any operation that would produce a NaN result
  - **Inexact result**
    - This is triggered by rounding (e.g. 1/10 in decimal cannot be represented in binary)
      - Not considered to be a serious exception (happens frequently), but can occur in combination with others

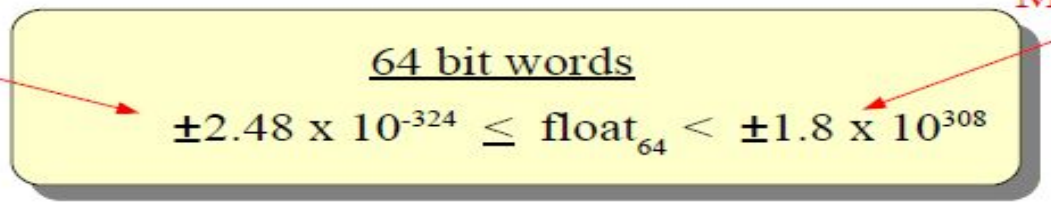
Overflow

- Recall when adding integers result can exceed the bit space available: an *overflow*
- Floating point arithmetic also can suffer from overflow
- When this occurs the floating point will give the result a value: an appropriately signed infinity
- Overflow is a potentially more serious error than underflow and is the frequent cause of crashes
  - An infinite result is found and then some attempt is made to manipulate it propagating an infinity through the calculation

Precision	Largest -ve value	Largest +ve value
Single	-3.4×10^{38}	3.4×10^{38}
Double	-1.7×10^{308}	1.7×10^{308}
Quad	-1.2×10^{4932}	1.2×10^{4932}

Underflow

- We've met underflow in the context of denormalized numbers
  - Result of calculation producing an answer that is smaller than that can be represented by normalized numbers
- Not as serious a problem as overflow, but in the absence of denormalized numbers there is a noticeable "hole" in the number line
  - Early Cray's didn't support denormalized numbers and frequently programs would show divide by zero problems
    - Solution is to add a small error bound to your result
- MINVAL ... MAXVAL



- If a number **x** is larger than the MAXVAL, an overflow occurs
- If a number **x** is smaller than the MINVAL, an underflow occurs

The resulting value may be a 0, INF, NINF, or NaN (not a number).

Arbitrary precision: Avoids underflow and overflow

- At a cost: multiplication of two numbers is naively an order n^2 operation (every digit of x must multiply every digit of y)
  - Remember we have to do this all in software – no hardware support
- Faster algorithms have been developed:
  - Karatsuba algorithm
  - Toom-Cook
  - Fourier transform methods

Check out David Bailey's page: http://crd.lbl.gov/~dhbailey

Aside: Experimental Mathematics

- Proofs of formulae, such as integrals or infinite sums are frequently difficult or impossible
- Can we show by numerical analysis that a given formula is very close to the exact answer?
  - Sufficiently close that the difference can be "ignored"

For example, the below result is known to be true to 20,000 digits of precision:

$$\frac{24}{7\sqrt{7}}\int_{\pi/3}^{\pi/2}\log\left|\frac{\tan t+\sqrt{7}}{\tan t-\sqrt{7}}\right|dt=\sum_{n=0}^{\infty}\left[\frac{1}{(7n+1)^2}+\frac{1}{(7n+2)^2}-\frac{1}{(7n+3)^2}+\frac{1}{(7n+4)^2}-\frac{1}{(7n+5)^2}-\frac{1}{(7n+6)^2}\right]$$

This is a useful result in quantum field theory, and it required almost an hour of computation on over 1000 processors.

Dr Jonathan Borwein at Dalhousie is a world leader in this emerging field.

Errors in Computational Science

- In the broadest sense there are 3 possible types of error
  - **Computational** errors
    - Discretization error
      - Conversion to a finite representation causes problems
    - Roundoff errors
    - Programming errors
  - **Data** errors
    - In accuracies in input data
  - **Modelling** errors
    - Underlying assumptions are incorrect

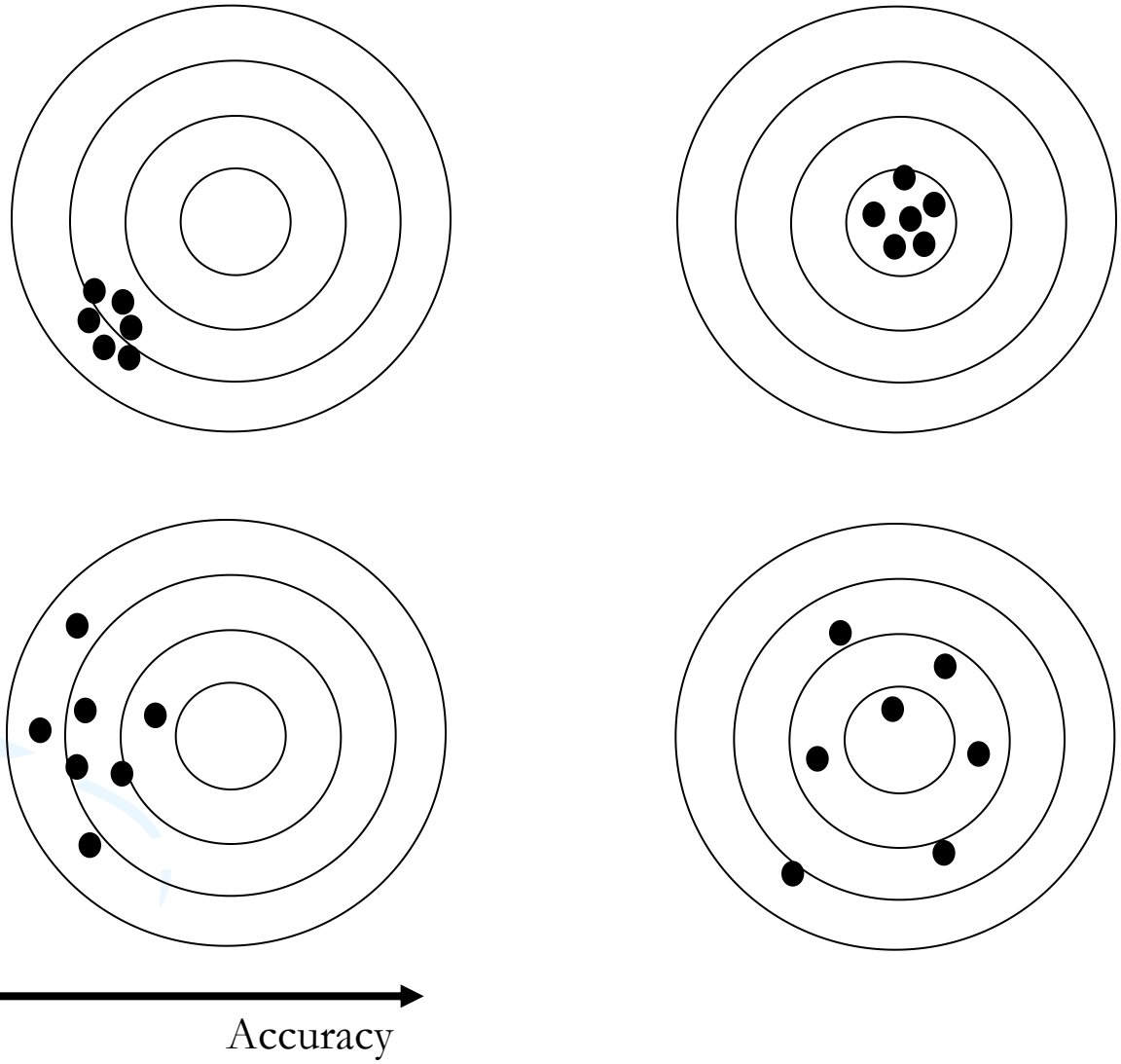
We'll look at all of these

Numerical errors are a serious issue…

On June 4, 1996, an Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development costing $7 billion. The problem was a software error (overflow) in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer.

On August 23,1991, the first concrete base structure for the Sleipner A platform sprang a leak and sank under a controlled ballasting operation during preparation for deck mating in Gandsfjorden outside Stavanger, Norway. The post accident investigation traced the error to inaccurate finite element approximation of the linear elastic model of the tricell (using the popular finite element program NASTRAN). The shear stresses were underestimated by 47% leading to insufficient design. In particular, certain concrete walls were not thick enough.

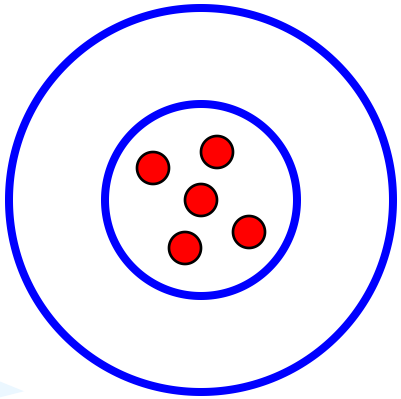
Recap: Accuracy versus Precision

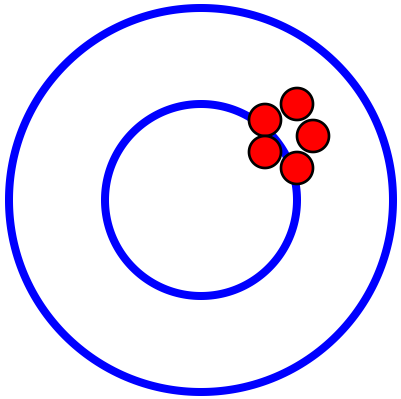


Accuracy and precision

- Accuracy - How closely a measured or computed value agrees with the true value
- Precision - How closely individual measured or computed values agree with each other

More Accurate





More Precise

- Accuracy is getting all your shots near the target.
- Precision is getting them close together.

Humour: Accuracy and Precision

"There's no sense in being precise when you don't even know what you're talking about."

J. von Neumann

Errors – a few definitions

- Given a value $\hat{w}$ that is an approximation to w, we can state the errors as
- Absolute error: $E_{abs}(\hat{w})=|w-\hat{w}|$
  - The absolute error between a floating point number and the number it represents is $\propto w$
  - This is not true for fixed point arithmetic
- Relative error: $E_{rel}(\hat{w})=|w-\hat{w}|/|w|$
  - Defines the number of significant figures in the answer

Rounding

- Rounding is an important operation in floating point calculations
  - Important when reduce precision (from double to single say)
  - Important when we perform fp operations due to shifting of the mantissa and exponents
- Rounding, as we learn in high school, has a bias
  - Any series of data containing .5 is biased upwards by applying upward rounding
- IEEE 754 contains 4 different rounding modes

Banker's rounding (round to even)

- Very similar to upwards rounding except you always round to nearest even number when the last digit is .5
  - e.g. 2.5→2 (not 3), 3.5 → 4
  - Should produce a more or less even distribution of truncation differences
- Why not round to odd?
  - You cannot round to zero in this scheme
- Round to even is standard rounding mode in IEEE 754
  - It is also referred to as unbiased rounding

Roundoff: Examples from finance

- Compare using integers to floating point representation

- Suppose you invest $1000.00 at 5% compounded daily for 1 year
  - After 1 year you should have $1051.27
  - Rounding to nearest penny each day give $1051.10
  - Rounding down would give $1049.40

Anyone remember Superman III?

Other rounding modes

- Rounding toward zero
  - This is the equivalent of just truncating
    - 1.01 → 1.0, whereas -1.01 → -1.0 (both are close to zero)
- Rounding towards positive INF
  - Example: -1.1 → -1.0 and 1.1 → 2.0
- Rounding towards negative INF
  - Example: -1.1 → -2.0 and 1.1 → 1.0
- The latter two options are important in *interval arithmetic*, where a number is represent by a pair of limits, e.g. x is represented by $[x_1,x_2]$ and $[x_1,x_2]+[y_1,y_2]=[x_1+y_1,x_2+y_2]$
  - x_1+y_1 would be rounded toward –INF, x_2+y_2 toward +INF

Floating point nomenclature

- As discussed, because of the finite nature of the representation, the floating point version of x may not actually be the same as x
  - $fl(x) \approx x$
- Under rounding the separation between x and $fl(x)$ must obey some minimum factor
  - $fl(x)=x(1+e)$, where $|e|<u$ (u is the *unit roundoff* from the previous lecture)
  - We can thus see where the definition of u comes from: set x to be 1, then roundoff is limited by ½ the spacing of the smallest difference between numbers

Roundoff Error example: adding floats

$$\begin{array}{r} 100000 \times 2^{10} \\ + \quad 123456 \times 2^{8} \\ \hline \end{array}$$

$$\begin{array}{r} 100000 \times 2^{10} \\ + \quad 123456 \times 2^{8} \\ \hline 101234.56 \times 2^{10} \end{array}$$ exact

$$\begin{array}{r} 100000 \times 2^{10} \\ + \quad 012345 \times 2^{9} \\ \hline \end{array}$$ exponential shifting

$$\begin{array}{r} 100000 \times 2^{10} \\ + \quad 001234 \times 2^{10} \\ \hline \end{array}$$ exponential shifting

$$\begin{array}{r} 100000 \times 2^{10} \\ + \quad 001234 \times 2^{10} \\ \hline 101234 \times 2^{10} \end{array}$$ exponential shifting

low-order bits are lost

Round-Off Error

Example: A Simple Sum

$$S = \sum \frac{1}{n}$$

$$S_{up} = \sum_{n=1}^{N} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$$

round-off error

$$S_{down} = \sum_{n=N}^{1} \frac{1}{n} = \frac{1}{N} + \frac{1}{N-1} + \frac{1}{N-2} + \ldots + 1$$

$$S_{up} \neq S_{down}$$

more precise

computational value → $x_c = x(1+\epsilon_x)$ $\quad |\epsilon_x| \le \epsilon$

true value

Subtractive Cancelation

$a = b - c \quad \rightarrow \quad a_c = b_c - c_c \qquad a_c = a(1+\epsilon_a)$

$$a(1+\epsilon_a) = b(1+\epsilon_b) - c(1+\epsilon_c)$$

$$\epsilon_a = \epsilon_b b/a - \epsilon_c c/a \; \boxed{-1 + (b-c)/a}$$

-1 + 1 = 0

$$\boxed{\epsilon_a = \epsilon_b b/a - \epsilon_c c/a}$$

if a is small then $b \approx c$

$$\epsilon_a \approx b/a(|\epsilon_b| + |\epsilon_b|) \qquad |\epsilon_i| \le \epsilon$$

If you subtract two large numbers and end up with a small one, The result will be less significant.

Multiplicative Errors

$$a = b * c \quad \rightarrow \quad a_c = b_c * c_c$$

$$a_c = a(1+\epsilon_a)$$

$$a(1+\epsilon_a) = b(1+\epsilon_b) * c(1+\epsilon_c)$$

$$1+\epsilon_a = 1+\epsilon_b+\epsilon_c+\epsilon_b\epsilon_c \rightarrow 0$$

$$\boxed{\epsilon_a = \epsilon_b + \epsilon_c}$$

Since ϵ_b and ϵ_c can have opposite signs the total error can be larger or smaller than the individual errors, but the overall distribution of errors is larger

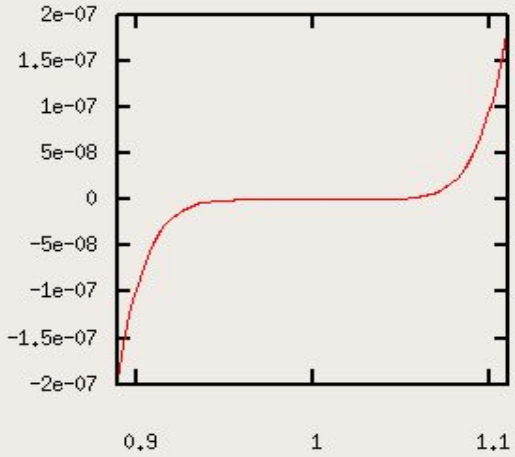
Floating Point numbers are stored as binary fractions and not decimal fractions

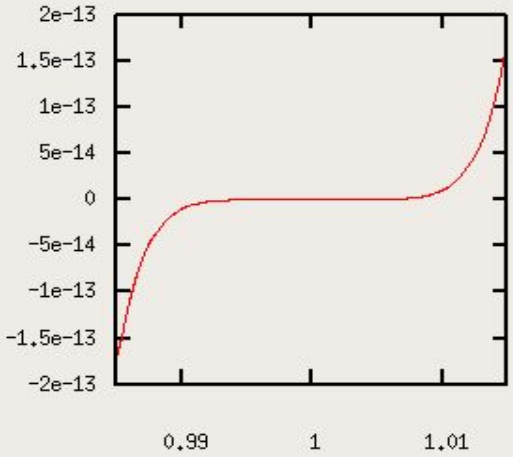
Example: 0.1254 = b0.001
0.125 = 1/10 + 2/100 + 5/1000
b0.001 = 0/2 + 0/4 + 1/8

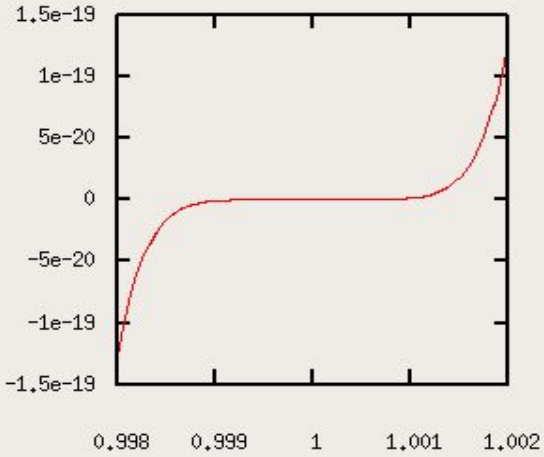
Most decimal fractions cannot be represented exactly as binary fractions

Zoom in on $(x-1)^7$

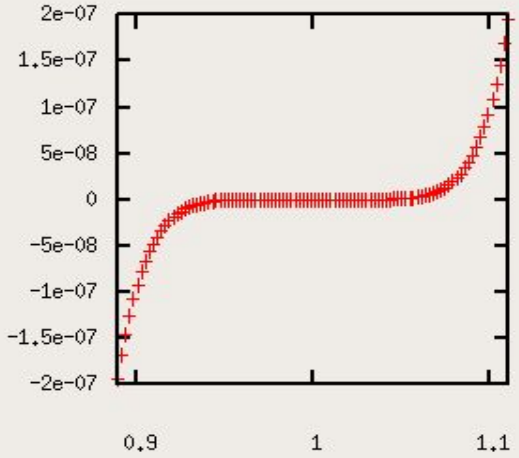
Both calculations done at double precision

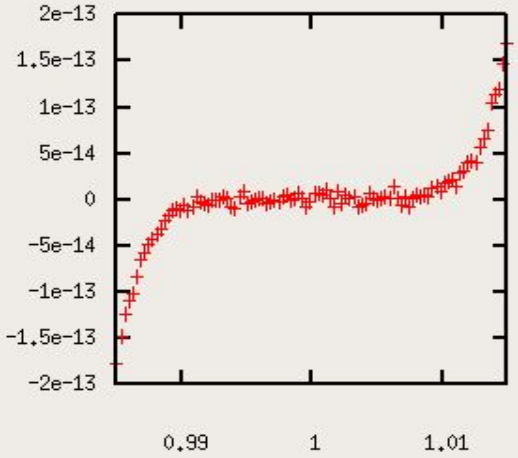


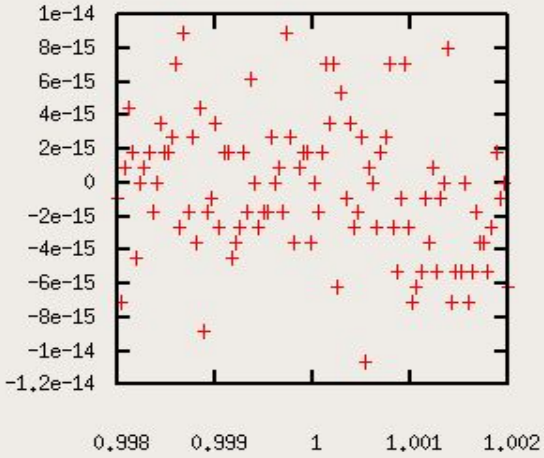




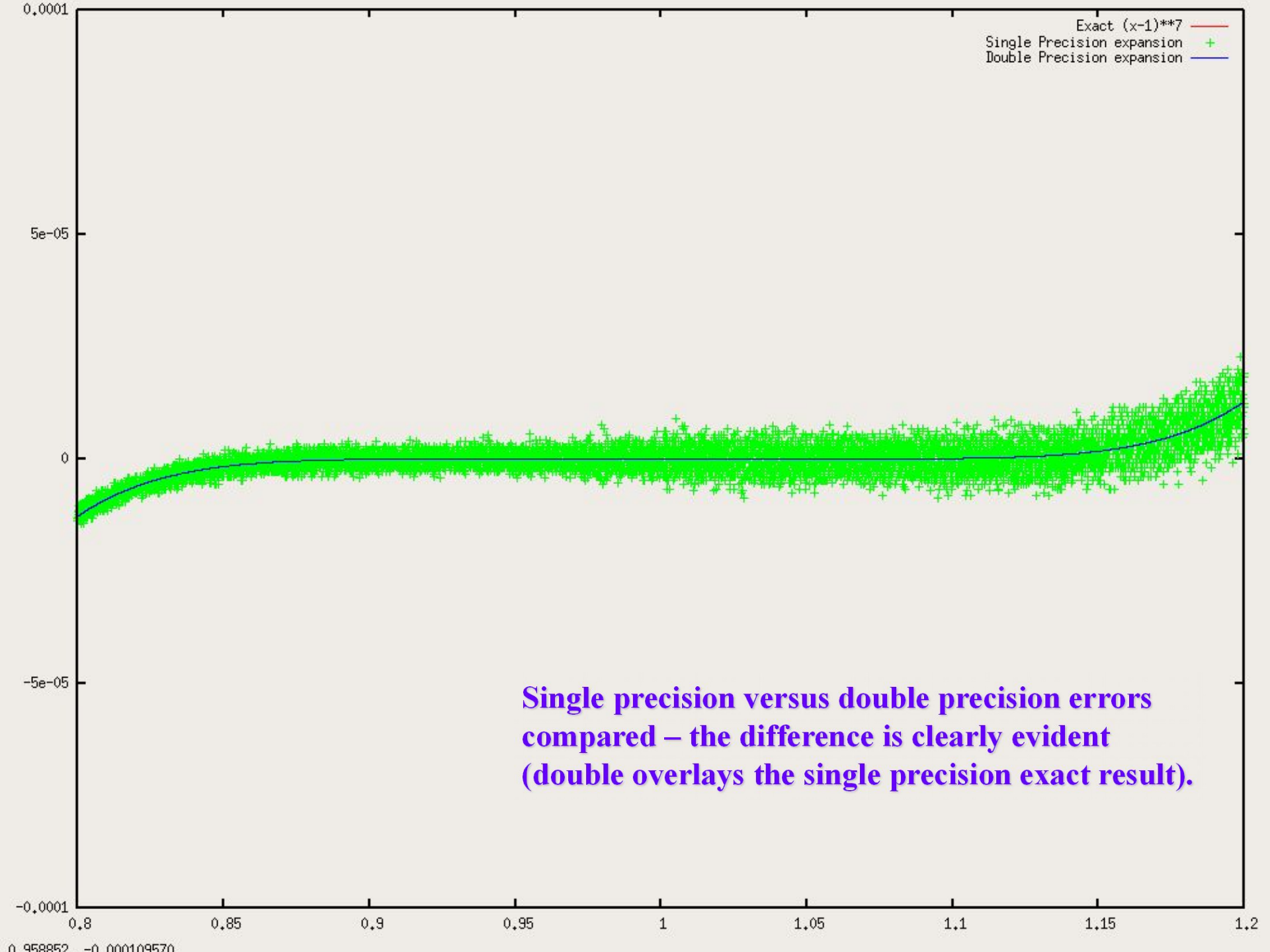
Zoom in on $x^7-7x^6+21x^5-35x^4+35x^3-21x^2+7x-1$ (expansion of $(x-1)^7$)







Error here is far larger than the expected machine epsilon



Single precision versus double precision errors compared – the difference is clearly evident (double overlays the single precision exact result).

Mathematical consequences

- Floating point calculations are commutative, for two numbers A,B and operator $\oplus$, where $\oplus$ is either addition or multiplication
  - $A \oplus B = B \oplus A$
- However, because of the rounding step they are not associative
  - $(A \oplus B) \oplus C \neq A \oplus (B \oplus C)$
- They are also not distributive
  - $(A+B) \times C \neq A \times C + B \times C$

Equating floating point numbers

- Asking whether $A=B$ in a floating point system is extremely problematic
- Rounding makes this an uncertain operation at best
- Better to ensure $|A-B|<d$, where d is a fixed tolerance that you can decide on
  - Tailor the precision you are using and the size of delta appropriately

Cancellation error

- When subtracting almost equal numbers you can lose precision
  - Two mantissa's that have similar values will cancel out the left most values on subtraction
- Catastrophic cancellation can occur when two numbers are subtracted that already have rounding errors of their own
- Consider b^2-$4ac$, where b=3.34, a=1.22, c=2.28 (accurate to within ½ the unit of the last place)
  - Exact answer is 0.0292
  - b^2 rounds to 11.2, $4ac$ rounds to 11.1 so the computed answer is .1 – a large error compared to the exact result!
  - |.1-0.0292|/0.0292=2.424 a 240% relative error!
- If $b^2 >> 4ac$ then $fl(-b+\sqrt{b^2-4ac})$ will be subject to a large error because you are subtracting two similar numbers

The standard quadratic formula is not a stable computational algorithm

Improving the quadratic formula

- There is an easy way around this though, the "poor" root can be rewritten in a better behaved form

$$\frac{-b-\sqrt{b^2-4ac}}{2a}=\frac{2c}{-b+\sqrt{b^2-4ac}}$$

$$\frac{-b+\sqrt{b^2-4ac}}{2a}=\frac{2c}{-b-\sqrt{b^2-4ac}}$$

Just a matter of multiplying general solution by $\frac{-b\pm\sqrt{b^2-4ac}}{-b\pm\sqrt{b^2-4ac}}$ and expanding terms

Mixing precisions

- It is tempting to believe a calculation mixing single and double precision values would be more accurate
  - This isn't the case – round-off in the single precision variables will still be present
- Ensure you work to the highest precision possible when necessary
  - Make sure all constants are set in the highest precision for example

Discretization error

- Thus far we have concentrated on roundoff error as the main source of errors in calculations

- Discretization error occurs whenever we try to model continuous functions with a set of discrete points
  - It is not the same as roundoff error which is caused by finite representations of numbers
  - Discretization error would exist even with infinite precision

We'll come back to discretization error later

Scaling: a technique that helps avoid underflow and overflow

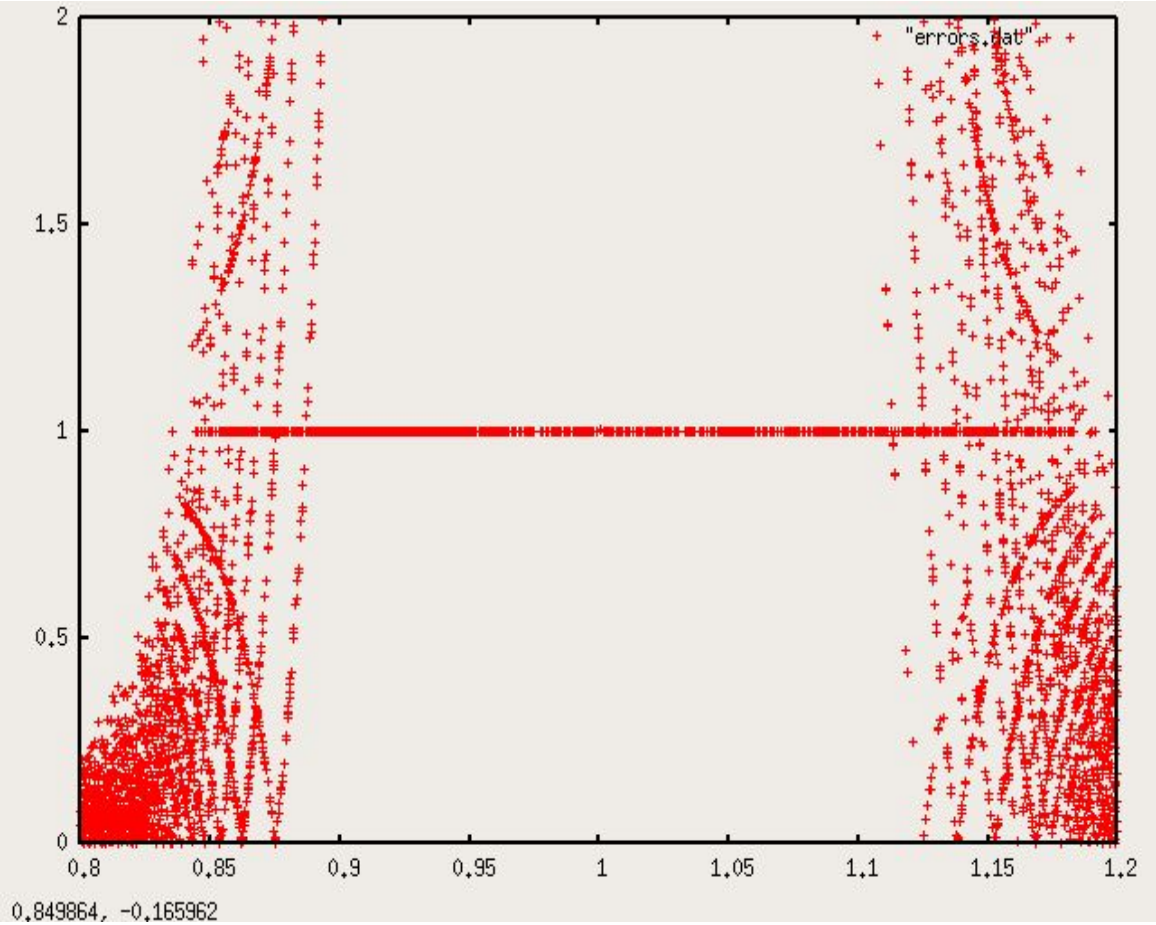
- Suppose we have two numbers $a=1.0\times10^{30}$, $b=1.0$, and we wish to compute $\sqrt{a^2+b^2}$
  - Without any alteration the result is +INF since the a^2 calculation overflows
- However, we can scale both these numbers by $1./max(a,b)$ and multiply the result by $max(a,b)$

  Then
  - $a \rightarrow a' = a/1.0\times10^{30}=1.$
  - $b \rightarrow b' = b/1.0\times10^{30}=1.0\times10^{-30}$

  Result is given by $1.0\times10^{30}\times\sqrt{a'^2+b'^2}=1.0\times10^{30}$
- Also works for underflow since numbers are scaled up

Propagation of roundoff

- Roundoff errors grow through a calculation, but the resulting errors they produce are NOT random!
  - Rounding is deterministic
- Graph shows the distribution of relative errors in the $(x\text{-}1)^7$ expansion



A note on stability of algorithms

- When you develop your own algorithms you need to ensure you are using a numerically stable approach
  - Be careful about the steps in your calculation, and ensure you look out for divide by small numbers, and cancellation issues

- Note: while roundoff produces one form of instability, some numerical methods are inherently unstable regardless of roundoff
  - We'll look at this more when we look at differential equations

Summary

- IEEE 754 contains a number of special numbers, 0,INF,NaN
- There are 5 exceptions, **invalid operation**, **divide by zero**, **overflow**, **underflow** and **inexact result**
  - Suitable precautions can reduce the probability of these exceptions being raised
- Errors are unavoidable in floating point calculations
  - Commutativity remains, but associativity and distributivity are lost
  - Steps can be taken to reduce the impact of roundoff, but you must be aware that will occur

Exercise 2

- For the quadratic formula (b=3.34, a=1.22, c=2.28), please compute the roots, and watch the Cancellation error.

- Please calculate $(x\text{-}1)^7$, and
  $x^7\text{-}7x^6\text{+}21x^5\text{-}35x^4\text{+}35x^3\text{-}21x^2\text{+}7x\text{-}1$ (expansion of $(x\text{-}1)^7$), in single, double, quad.

  zoom in the plots of these calculations.

The standard quadratic formula is not a stable computational algorithm

Homwork 3： 03/20/2019

Problem 1: The root of of quadratic equation

$$x^2 - bx + 1 = 0$$

is $x_1 = \dfrac{b+r}{2}$; $x_2 = \dfrac{b-r}{2}$ *where* $r = \sqrt{b^2 - 4}$

If *b* is large, *r* is close to *b*, *there is* potential for greater error!

1) Try *b* = 100, you will watch 2% relative error for the second root

2) Please try other large *b(e.g., 1000, 10000, 100000, etc)*, you will watch much higher relative error, please table your results

Now rationalize: $x_2 = \dfrac{(b-r)}{2}\dfrac{(b+r)}{(b+r)} = \dfrac{2}{b+r}$

Redo all these works, do you see higher relative error or not? Please explain why?

Try $b = 97$ $\quad x^2 - 97x + 1 = 0$

$(r = 96.9794)$

x_2 (3 sig. figs.)

exact:	0.01031
standard:	0.01050
rationalized:	0.01031

Corresponding to “cancellation, critical arithmetic”

Problem 2: Please calculate $(x\text{-}1)^9$ and expansion of $(x\text{-}1)^9$, in single, double, quad.

1) zoom in the plots of these calculations. (x: 0.7~1.3)

2) Draw the distribution of relative errors in the $(x\text{-}1)^9$ expansion

3) Please compute in the expansion of $(x\text{-}1)9$ in **Horner's Method** , tell us what you see? and explain why?

Horner's Method is a way of expressing a polynomial **f(x)** which eliminates all exponentiations. By eliminating exponentiations, we eliminate re–doing some calculations.

Constant Function: $f(x) = c_0$

Linear Function: $f(x) = c_1 x + c_0$

Quadratic Function: $f(x) = c_2 x^2 + c_1 x + c_0$

$$f(x) = (c_2 x + c_1) x + c_0$$

In general,

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + c_{n-2} x^{n-2} + \ldots + c_2 x^2 + c_1 x + c_0$$

$$f(x) = (c_n x^{n-1} + c_{n-1} x^{n-2} + c_{n-2} x^{n-3} + \ldots + c_2 x + c_1) x + c_0$$

$$f(x) = ((c_n x^{n-2} + c_{n-1} x^{n-3} + c_{n-2} x^{n-4} + \ldots + c_2) x + c_1) x + c_0$$

$$f(x) = (\ldots ((c_n x^2 + c_{n-1} x + c_{n-2}) x + \ldots + c_2) x + c_1) x + c_0$$

$$f(x) = ((\ldots ((c_n x + c_{n-1}) x + c_{n-2}) x + \ldots + c_2) x + c_1) x + c_0$$