Unit 0

Numerical Computation

Numerical computation

- digital representation of numbers
- floating point arithmetic
- implications for routine calculations
- sources of error
- introduction to Matlab

Integers

- binary representation
  - a *bit* is 0 or 1
  - 8 bits = 1 *byte*
  - 2 bytes = *hex* digit
- $27_{10} = 2^4 + 2^3 + 2^1 + 2^0 = [0001\ 1011]_2 = [1B]_{16}$
- integers have an exact binary representation
- typical implementation allocates 16 bit integer size
- 32 bit integers also available

Integers

- limitation on unsigned 16 bit integer magnitude is $2^{16} = 65536$
- how to store negative integers without a sign bit?
  - use twos complement notation
  - to represent a negative value complement all the bits and add one
  - equivalently [for n bits] subtract all values from 2^n before storing
- range of [-32768,32767] for integers
  - one representation is required for zero so asymmetric

Non-integers

- scientific notation: 1234.56 = 1.23456 E+03
  - *sign* +
  - *mantissa* (*significand*) 1.23456
  - *exponent* +3
- floating point representation allocates a fixed number of bits to each non-integer
- requires a convention on how the bits are used for sign, mantissa, and exponent
- a *normalized* binary FPV could have a mantissa of 0 or 1 by convention

Non-integers

- a *single precision* floating point number
  - 32 bits = 1 sign bit + 23bit mantissa + 8bit exponent
- a *double precision* floating point number
  - 64 bits = 1 sign bit + 52bit mantissa + 11bit exponent
- a *normalized* binary FPV assumes the mantissa always represents a fixed decimal location
  - one convention is use 1.bbbbb
  - first digit is the J-bit and can be assumed
- Matlab default: all calculations use double precision floating point arithmetic
  - exact integer arithmetic is also available

Non-integers

- floating point mantissa is expressed in powers of 1/2
  - $(1/2)^0 = 1$ not used [fixed J-bit assumed]
  - $(1/2)^1 = 0.5$
  - $(1/2)^2 = 0.25$
  - $(1/2)^3 = 0.125$
  - $(1/2)^4 = 0.0625$
- to find the binary representation for a decimal number
  - subtract successive powers of 1/2 until reduced to zero or you run out of bits
  - $0.8125_{10} = 0.5 + 0.25 + 0.0625 = (1/2)^1 + (1/2)^2 + (1/2)^4 = [0.1101]_2$

Roundoff

- many [most] exact decimal mantissas cannot be represented exactly as binary mantissas
  - example $0.1 = [0.000110011...]_2$
- exact floating point representation is only possible for
  - integers less than 2^{52} or
  - numbers with 15 [decimal] bit mantissa an exact sum of 1/2 powers
- all other decimal real numbers must be represented in binary as approximations
- the limited number of mantissa bits limits precision = number of significant bits in the approximation
  - the floating point number line is full of holes.....
  - *eps* ~ 2.2204×10^{-16} is smallest machine value so that 1.0+eps is different from 1.0 [called *machine precision*]

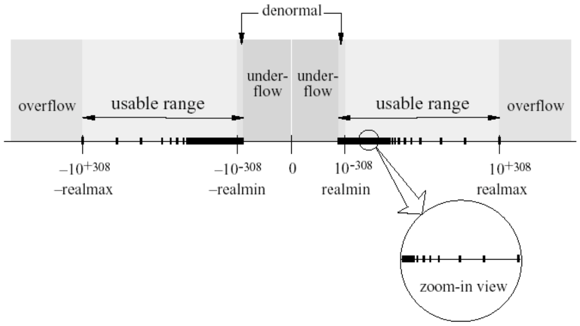
Overflow and underflow

- the number of exponent bits limits upper and lower floating point magnitudes
- how to represent the exponent sign?
  - add a bias value to exponents
  - single precision adds $127=2^7-1$ bias
  - double precision adds $1023=2^{10}-1$ bias
- largest exponent possible is 1023
- so largest floating point magnitude is about $2^{1023} \sim 8.99\times10^{307}$
- special values are used to represent maximum and minimum floating point numbers for a given computer design
  - *realmax* ~ 10^{308}
  - *realmin* ~ 10^{-308}

Overflow and underflow

- floating point values < realmin cause *underflow*
  - handled differently according to computer design
  - may be replaced by zero
  - some computers use *denormalized* FPVs to handle some underflow values
  - mantissa bits are lost so precision is reduced
- floating point values > realmax cause *overflow*
  - often replaced by a special value called *infinity*
- special machine values can be used in calculations with the anticipated results
- all the above applies by symmetry to negative floating point numbers as well

The floating point number line



Floating point arithmetic

- limited range and precision of floating point numerical values has implications for routine calculation
- result of an arithmetic operation on two FPVs might not be representable as an FPV
- rounding error can lead to unrecoverable loss of significant bits

Floating point arithmetic: bad things to avoid

- effects of roundoff errors accumulate slowly but....
- catastrophic cancellation error is
  - caused by a single operation when...
  - ...subtracting two nearly equal values or
  - ...adding two very different values
  - critical loss of significant digits can occur in routine calculations
- roundoff cannot be avoided so the solution is to improve the algorithms
- computer calculations are not necessarily organized in the same order as hand calculations
  - re-arranged quadratic formula
  - nested evaluation of polynomial expressions

Floating point arithmetic: another implication

- floating point numbers that are supposed to be equal may not be equal due to roundoff
- so floating point comparisons should always involve ‘close enough’ and NEVER ‘equals’

- how is ‘close enough’ quantified?

Close enough?

- x_T is the true value of a quantity x and
- x_A is a computed value of x
- two approaches to quantifying error....
- the *absolute error* is

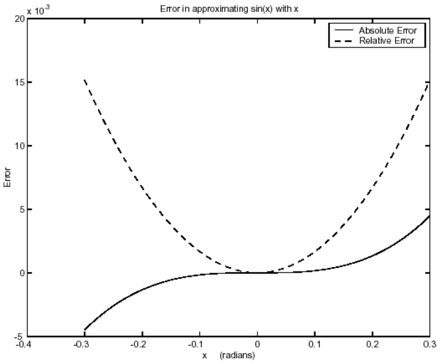
  $$\text{error}(x_A) = x_T - x_A$$
- the *relative error* is

  $$\text{rel}(x_A) = (x_T - x_A) / x_T$$

- when comparing two FPVs ask: ‘is $|x-y| < tol$?
  - the *tolerance* may be chosen to be absolute or relative according to problem specifics

---

Comparing sin x and x



Truncation error

- terminating an iteration process results in a *truncation error* or *discretization error*
- $f_T = f_A$ + truncation error
- caused by the algorithm not the computer
- size of truncation error in evaluating f(x) depends on x and the number of terms used
  - e.g. for large x the Taylor series for sin x converges more slowly than for x ~ 0

- both roundoff and truncation errors are present in numerical calculations

---

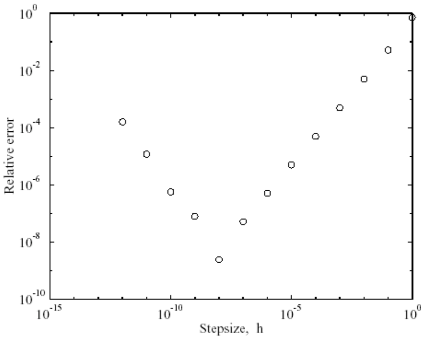
Truncation vs roundoff error

- series expansion of $f(x) = e^x$
- $T_k = x^k/k!$
- $S_k = S_{k-1} + T_k$
- what happens when the series is terminated after k terms?
- Matlab example:
  - *expseriesplot(x,tol,k)* plots abs error $|S_k - \exp(x)|$
  - $|x| >> 1$ abs error increases first as numerator terms grow more slowly than the factorial
  - for x = -10 factorial begins to dominate the error at 10 terms
  - truncation error decreases with increasing number of terms
  - eventually you get caught by roundoff error when you reach eps....no further change in S_k

Truncation vs roundoff error

Roundoff error dominates

Truncation error dominates



Evaluation of f′(x) using finite differences; f(x) = exp(x)