Unavoidable Errors in Computing

Gerald W. Recktenwald
Department of Mechanical Engineering
Portland State University
gerry@me.pdx.edu

The latest version of this PDF file, along with other supplemental material for the book, can be found at www.prenhall.com/recktenwald.

Version 0.951   December 8, 2001

Overview

- Digital representation of number
  - ▷ Size limits
  - ▷ Resolution limits
  - ▷ The floating point number line
- Floating point arithmetic
  - ▷ roundoff
  - ▷ machine precision
- Implications for routine computation
  - ▷ Use "close enough" instead of "equals"
  - ▷ loss of significance for addition
  - ▷ catastrophic cancellation for subtraction
- Truncation error
  - ▷ Demonstrate with Taylor series
  - ▷ Order Notation

What's going on here?

Spontaneous generation of an insignificant digit:

```
>> format long e  % display lots of digits
>> 2.6 + 0.2
ans =
        2.800000000000000e+00

>> ans + 0.2
ans =
        3.000000000000000e+00

>> ans + 0.2
ans =
        3.200000000000001e+00

>> 2.6 + 0.6
ans =
        3.200000000000000e+00
```

Bits, Bytes, and Words

base 10	conversion	base 2
1	$1 = 2^0$	0000 0001
2	$2 = 2^1$	0000 0010
4	$4 = 2^2$	0000 0100
8	$8 = 2^3$	0000 1000
9	$8 + 1 = 2^3 + 2^0$	0000 1001
10	$8 + 2 = 2^3 + 2^1$	0000 1010
27	$16 + 8 + 2 + 1 = 2^4 + 2^3 + 2^1 + 2^0$	$\underbrace{0001\ 1011}_{\text{one byte}}$

Digital Storage of Integers (1)

- Integers can be exactly represented by base 2
- Typical size is 16 bits
- $2^{16} = 65536$ is largest 16 bit integer
- $[-32768, 32767]$ is range of 16 bit integers in twos complement notation
- 32 bit and larger integers are available

Note: All standard mathematical calculations in MATLAB use floating point numbers. Describing binary storage of integers is a prelude to discussing the binary storage of non-integers.

Expert's Note: The built-in `int8`, `int16`, `int32`, `uint8`, `uint16`, and `uint32` classes are meant as a means of reducing data storage costs.

Digital Storage of Integers (2)

Let b be a binary digit, i.e. 1 or 0

$$(bbbb)_2 \iff |2^3|2^2|2^1|2^0|$$

The **rightmost bit** is the **least significant bit** (LSB)

The **leftmost bit** is the **most significant bit** (MSB)

Example:

$$\begin{aligned}(1001)_2 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 0 + 0 + 1 = 9\end{aligned}$$

Digital Storage of Integers (3)

Limitations:

- computers store values in memory with a fixed number of bits
- Limiting the number of bits limits the size of integer that can be represented

max 3 bit integer:	$(111)_2$	$= 4 + 2 + 1 = 7$	$= 2^3 - 1$
max 4 bit integer:	$(1111)_2$	$= 8 + 4 + 2 + 1 = 15$	$= 2^4 - 1$
max 5 bit integer:	$(11111)_2$	$= 16 + 8 + 4 + 2 + 1 = 31$	$= 2^5 - 1$
max n bit integer:			$= 2^n - 1$

Digital Storage of Non-integer Numbers (1)

- Use normalized scientific notation:

$$123.456 \quad \longrightarrow \quad 0.123456 \times 10^3$$

- Fixed number of bits are allocated to each number
  - ▷ *single precision* uses 32 bits per floating point number
  - ▷ *double precision* uses 64 bits per floating point number

- Total number of bits are split into separate storage for the mantissa and exponent
  - ▷ *single precision*: 1 sign bit, 23 bit mantissa, 8 bit exponent
  - ▷ *double precision*: 1 sign bit, 52 bit mantissa, 11 bit exponent

Digital Storage of Non-integer Numbers (2)

Numeric values with non-zero fractional parts are stored as **floating point numbers**.

All floating point values are represented with a normalized scientific notation.

Example:

$$12.3792 = \underbrace{0.123792}_{\text{mantissa}} \times 10^2$$

Digital Storage of Non-integer Numbers (3)

Floating point values have a fixed number of bits allocated for storage of the mantissa and a fixed number of bits allocated for storage of the exponent.

Two common precisions are provided in numeric computing languages

Precision	Bits for mantissa	Bits for exponent
Single	23	8
Double	53	11

Digital Storage of Non-integer Numbers (4)

A double precision (64 bit) floating point number can be schematically represented as

$$\overbrace{\underbrace{b}_{\text{sign}}\ \underbrace{bb\ldots\ldots bbb}_{\substack{\text{52 bit value}\\ \text{of mantissa}}}\ \underbrace{bbbbbbbbbbb}_{\substack{\text{11 bit exponent,}\\ \text{including sign}}}}^{64\ \text{bits}}$$

Digital Storage of Non-integer Numbers (5)

Floating Point mantissa expressed in powers of $\frac{1}{2}$

$$\left(\frac{1}{2}\right)^0 = 1 \qquad \text{not used}$$

$$\left(\frac{1}{2}\right)^1 = 0.5$$

$$\left(\frac{1}{2}\right)^2 = 0.25$$

$$\left(\frac{1}{2}\right)^3 = 0.125$$

$$\left(\frac{1}{2}\right)^4 = 0.0625$$

$$\vdots$$

Digital Storage of Non-integer Numbers (6)

Example: *Binary mantissa for* $x = 0.8125$

Apply Algorithm 5.1

k	2^{-k}	b_k	$r_k = r_{k-1} - b_k 2^{-k}$
0	NA	NA	0.8125
1	0.5	1	0.3125
2	0.25	1	0.0625
3	0.125	0	0.0625
4	0.0625	1	0.0000

Therefore, the binary mantissa for 0.8125 is (exactly) $(1101)_2$

Digital Storage of Non-integer Numbers (7)

Example: *Binary mantissa for* $x = 0.1$

Apply Algorithm 5.1

k	2^{-k}	b_k	$r_k = r_{k-1} - b_k 2^{-k}$
0	NA	NA	0.1
1	0.5	0	0.1
2	0.25	0	0.1
3	0.125	0	0.1
4	0.0625	1	0.1 - 0.0625 = 0.0375
5	0.03125	1	0.0375 - 0.03125 = 0.00625
6	0.015625	0	0.00625
7	0.0078125	0	0.00625
8	0.00390625	1	0.00625 - 0.00390625 = 0.00234375
9	0.001953125	1	0.0234375 - 0.001953125 = 0.000390625
10	0.0009765625	0	0.000390625
⋮	⋮		

Therefore, the binary mantissa for 0.1 is $(00011\,0011\,\ldots)_2$.

The decimal value of 0.1 cannot be represented by a finite number of binary digits.

Digital Storage of Non-integer Numbers (8)

Consequences

- Limiting the number of bits allocated for storage of the exponent means that there are upper and lower limits on the magnitude of floating point numbers
- Limiting the number of bits allocated for storage of the mantissa means that there is a limit to the precision (number of significant digits) for any floating point number.
- Most real numbers cannot be stored exactly (they do not exist on the floating point number line)
  - ▷ Integers less than 2^{52} can be stored exactly. Try

```
>> x = 2^51
>> s = dec2bin(x)
>> x2 = bin2dec(s)
>> x2-x
```

  - ▷ Numbers with 15 (decimal) digit mantissas that are the exact sum of powers of (1/2) can be stored exactly

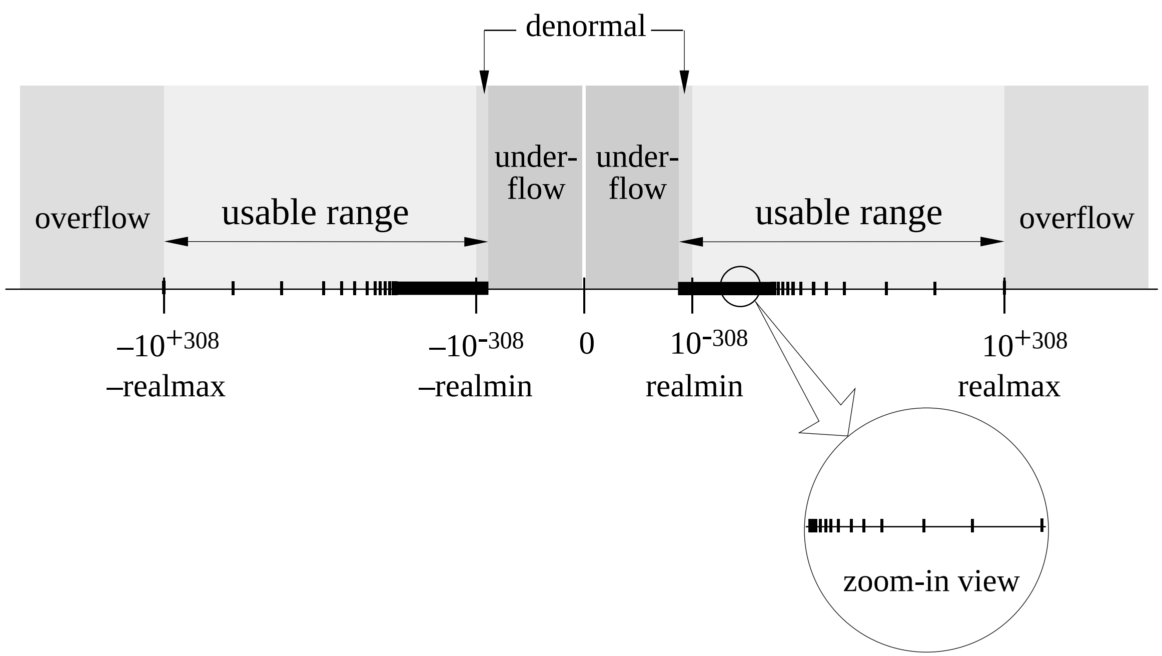
Floating Point Number Line

Compare floating point numbers to real numbers.

	Real numbers	Floating point numbers
Range	Infinite: arbitrarily large and arbitrarily small real numbers exist.	Finite: the number of bits allocated to the exponent limit the magnitude of floating point values
Precision	Infinite: There is an infinite set of real numbers between any two real numbers.	Finite: there is a finite number (perhaps zero) of floating point values between any two floating point values.

In other words: The floating point number line is a subset of the real number line.

Floating Point Number Line



Symbolic versus Numeric Calculation (1)

Commercial software for symbolic computation

- Derive$^{\mathrm{TM}}$
- MACSYMA$^{\mathrm{TM}}$
- Maple$^{\mathrm{TM}}$
- Mathematica$^{\mathrm{TM}}$

Symbolic calculations are exact. No rounding occurs because symbols can be manipulated without substituting numerical values.

Symbolic versus Numeric Calculation (2)

Example: *Evaluate* $f(\theta) = 1 - \sin^2\theta - \cos^2\theta$

Numerical computation in MATLAB:

```
>> theta = 30*pi/180;     %  must assign theta before it is used
>> f = 1 - sin(theta)^2 - cos(theta)^2
f =
  -1.1102e-16
```

`f` is close to, but not exactly equal to zero because of *roundoff*. Also note that `f` is a single value, not a formula.

Symbolic versus Numeric Calculation (3)

Symbolic computation using the Symbolic Math Toolbox in MATLAB

```
>> t = sym('t')     %  declare t as a symbolic variable
t =
t

>> f = 1 - sin(t)^2 - cos(t)^2  %  create a symbolic expression
f =
1-sin(t)^2-cos(t)^2

>> simplify(f)    %  ask Maple to make algebraic simplifications
f =
0
```

In the symbolic computation, `f` is exactly zero for any value of `t`. There is no roundoff error in symbolic computation.

Numerical Arithmetic

Numerical values have limited range and precision. Values created by adding, subtracting, multiplying, or dividing floating point values will also have limited range and precision.

Quite often, the result of an arithmetic operation between two floating point values cannot be represented as another floating point value.

Integer Arithmetic

Operation	Result
$2 + 2 = 4$	integer
$9 \times 7 = 63$	integer
$\frac{12}{3} = 4$	integer
$\frac{29}{13} = 2$	exact result is not an integer
$\frac{29}{1300} = 0$	exact result is not an integer

Floating Point Arithmetic

Operation	Result
$2.0 + 2.0 = 4$	floating point value is exact
$9.0 \times 7.0 = 63$	floating point value is exact
$\dfrac{12.0}{3.0} = 4$	floating point value is exact
$\dfrac{29}{13} = 2.230769230769231$	floating point value is approximate
$\dfrac{29}{1300} = 2.230769230769231 \times 10^{-2}$	floating point value is approximate

Floating Point Arithmetic in Matlab (1)

```
>> format long e
>> u = 29/13
u =
     2.230769230769231e+00

>> v = 13*u
v =
    29
>> v-29
ans =
     0
```

Two rounding errors are made in sequence: (1) during computation and storage of `u`, and (2) during computation and storage of `v`. Fortuitously, the combination of rounding errors produces the exact result.

Floating Point Arithmetic in MATLAB (2)

```
>> x = 29/1300
x =
     2.230769230769231e-02

>> y = 29 - 1300*x
y =
    3.552713678800501e-015
```

In exact arithmetic, the value of y should be zero.

The roundoff error occurs when x is stored. Since $29/1300$ cannot be expressed with a finite sum of the powers of $1/2$, the numerical value stored in x is a truncated approximation to $29/1300$.

When y is computed, the expression 1300*x evaluates to a number slightly different than 29 because the bits lost in the computation and storage of x are not recoverable.

Roundoff in Quadratic Equation (1)

(See **Example 5.3** in the text)

The roots of

$$ax^2 + bx + c = 0 \tag{1}$$

are

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

Consider

$$x^2 + 54.32x + 0.1 = 0 \tag{3}$$

which has the roots (to eleven digits)

$$x_1 = 54.3218158995, \qquad x_2 = 0.0018410049576.$$

Note that $b^2 \gg 4ac$

$$b^2 = 2950.7 \gg 4ac = 0.4$$

Roundoff in Quadratic Equation (2)

Compute roots with four digit arithmetic

$$
\begin{aligned}
\sqrt{b^2 - 4ac} &= \sqrt{(-54.32)^2 - 0.4000} \\
&= \sqrt{2951 - 0.4000} \\
&= \sqrt{2951} \\
&= 54.32
\end{aligned}
$$

Use $x_{1,4}$ to designate the first root computed with four-digit arithmetic:

$$
\begin{aligned}
x_{1,4} &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} && (i) \\
&= \frac{+54.32 + 54.32}{2.000} && (ii) \\
&= \frac{108.6}{2.000} && (iii) \\
&= 54.30 && (iv)
\end{aligned}
$$

Correct root is $x_1 = 54.3218158995$. Four digit arithmetic leads to 0.4 percent error in this example.

Roundoff in Quadratic Equation (3)

Using four-digit arithmetic the second root, $x_{2,4}$, is

$$
\begin{aligned}
x_{2,4} &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \\
&= \frac{+54.32 - 54.32}{2.000} && (i) \\
&= \frac{0.000}{2.000} && (ii) \\
&= 0, && (iii)
\end{aligned}
$$

An error of 100 percent!

The poor approximation to $x_{2,4}$ is caused by roundoff in the calculation of $\sqrt{b^2 - 4ac}$. This leads to the subtraction of two equal numbers in line (i).

Roundoff in Quadratic Equation (4)

A solution: rationalize the numerators of the expressions for the two roots:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \left(\frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} \right) \tag{4}$$

$$= \frac{2c}{-b - \sqrt{b^2 - 4ac}}, \tag{5}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \left(\frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} \right) \tag{6}$$

$$= \frac{2c}{-b + \sqrt{b^2 - 4ac}} \tag{7}$$

Roundoff in Quadratic Equation (5)

Now use Equation (7) to compute the troublesome second root with four digit arithmetic

$$
\begin{aligned}
x_{2,4} &= \frac{2c}{-b + \sqrt{b^2 - 4ac}} \\
&= \frac{0.2000}{+54.32 + 54.32} \\
&= \frac{0.2000}{108.6} \\
&= 0.001842.
\end{aligned}
$$

The result is in error by only 0.05 percent.

The two formulations for $x_{2,4}$ are algebraically equivalent. The difference in the computed result is due to roundoff alone

Roundoff in Quadratic Equation (6)

Repeat the calculation of $x_{1,4}$ with the new formula

$$\begin{aligned} x_{1,4} &= \frac{2c}{-b - \sqrt{b^2 - 4ac}} \\ &= \frac{0.2000}{+54.32 - 54.32} && (i) \\ &= \frac{0.2000}{0} && (ii) \\ &= \infty. \end{aligned}$$

Limited precision in the calculation of $\sqrt{b^2 + 4ac}$ leads to a *catastrophic cancellation error* in step (i)

Roundoff in Quadratic Equation (7)

A robust solution is to use a formula that takes the sign of b into account in a way that prevents catastrophic cancellation.

The ultimate quadratic formula:

$$q \equiv -\frac{1}{2}\left[b + \mathrm{sign}(b)\sqrt{b^2 - 4ac}\right]$$

where

$$\mathrm{sign}(b) = \begin{cases} 1 & \text{if } b \geq 0, \\ -1 & \text{otherwise} \end{cases}$$

Then roots to quadratic equation are

$$x_1 = \frac{q}{a} \qquad x_2 = \frac{c}{q}$$

Roundoff in Quadratic Equation (8)

Summary

- Finite-precision causes roundoff in individual calculations
- Effects of roundoff accumulate slowly
- Subtracting nearly equal numbers leads to severe loss of precision. A similar loss of precision occurs when two numbers of very different magnitude are added.
- Since roundoff is inevitable, solution is to create better algorithms

Catastrophic Cancellation Errors (1)

For addition: The errors in

$$c = a + b \qquad \text{and} \qquad c = a - b$$

will be large when $a \gg b$ or $a \ll b$.

Consider $c = a + b$ with $a = x.xxx\ldots \times 10^0$,
$b = y.yyy\ldots \times 10^{-8}$, where x and y are decimal digits.
Assume for convenience of exposition that $z = x + y < 10$.

$$\begin{array}{cl}
 & \overbrace{\mathtt{x.xxx\ xxxx\ xxxx\ xxxx}}^{\text{available precision}} \\
+ & \mathtt{0.000\ 0000\ yyyy\ yyyy\ yyyy\ yyyy} \\
\hline
= & \mathtt{x.xxx\ xxxx\ zzzz\ zzzz}\ \underbrace{\mathtt{yyyy\ yyyy}}_{\text{lost digits}}
\end{array}$$

The most significant digits of a are retained, but the least significant digits of b are lost because of the mismatch in magnitude of a and b.

Catastrophic Cancellation Errors (2)

For subtraction: The error in

$$c = a - b$$

will be large when $a \approx b$.

Consider $c = a - b$ with

$$a = x.xxxxxxxxxxx1ssssss$$

$$b = x.xxxxxxxxxxx0tttttt$$

where x, y, s and t are decimal digits. The digits $sss\ldots$ and $ttt\ldots$ are lost when a and b are stored in double-precision, floating point format.

Catastrophic Cancellation Errors (3)

Evaluate $a - b$ in floating point arithmetic:

$$\begin{array}{rl} & \overbrace{\mathrm{x.xxx\,xxxx\,xxxx}}^{\text{available precision}}\,1 \\ - & \mathrm{x.xxx\,xxxx\,xxxx}\,0 \\ \hline = & 0.000\,0000\,0000\,1\ \ \underbrace{\mathrm{uuuu\,uuuu\,uuuu}}_{\text{unassigned digits}} \\ = & 1.\mathrm{uuuu\,uuuu\,uuuu} \times 10^{-12} \end{array}$$

The result has only one significant digit. Values for the uuuu digits are not necessarily zero. The *absolute* error in the result is small compared to either a or b. The *relative* error in the result is large because $ssssss - tttttt \neq uuuuuu$ (except by chance).

Catastrophic Cancellation Errors (4)

Summary

- Occurs in addition: $\alpha + \beta$ when $\alpha \gg \beta$ or $\alpha \ll \beta$
- Occurs in subtraction: $\alpha - \beta$ when $\alpha \approx \beta$
- Error caused by a single operation (hence the term "catastrophic") not a slow accumulation of errors.
- Can often be minimized by algebraic rearrangement of the troublesome formula. (Cf. improved quadratic formula.)

Machine Precision (1)

The magnitude of roundoff errors is quantified by *machine precision* ε_m.

There is a number, ε_m such that

$$1 + \delta = 1$$

whenever $\delta < \varepsilon_m$.

In exact arithmetic, ε_m is identically zero.

MATLAB uses double precision (64 bit) arithmetic. The built-in variable **eps** stores the value of ε_m.

$$\texttt{eps} = 2.2204 \times 10^{-16}$$

Machine Precision (2)

Algorithm for Computing Machine Precision

```
epsilon = 1;
it = 0;
maxit = 100;
while it < maxit
    epsilon = epsilon/2;
    b = 1 + epsilon;
    if b == 1
        break;
    end
    it = it + 1;
end
```

Implications for Routine Calculations

- Floating point comparisons should involve “close enough” instead of exact equality
- Terminate iterations when subsequent values are “close enough”.
- Express “close” in terms of
  - ▷ absolute difference, or
  - ▷ relative difference

Floating Point Comparison

Don't ask "is x equal to y".

```
if x==y       %  Don't do this
   ...
end
```

Instead ask, "are x and y 'close enough' in value"

```
if abs(x-y) < tol
   ...
end
```

Absolute and Relative Error (1)

"Close enough" can be measured with either absolute error or relative error, or both

Let

$$\alpha = \text{some exact or reference value}$$
$$\widehat{\alpha} = \text{some computed value}$$

Absolute error

$$E_{\text{abs}}(\widehat{\alpha}) = \left|\widehat{\alpha} - \alpha\right|$$

Relative error

$$E_{\text{rel}}(\widehat{\alpha}) = \frac{\left|\widehat{\alpha} - \alpha\right|}{\left|\alpha_{\text{ref}}\right|}$$

Often we choose $\alpha_{\text{ref}} = \alpha$ so that

$$E_{\text{rel}}(\widehat{\alpha}) = \frac{\left|\widehat{\alpha} - \alpha\right|}{\left|\alpha\right|}$$

Absolute and Relative Error (2)

Example: *Approximating* $\sin(x)$ *for small* x

Since

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

we can approximate $\sin(x)$ with

$$\sin(x) \approx x$$

for small enough $x < 1$

The absolute error in this approximation is

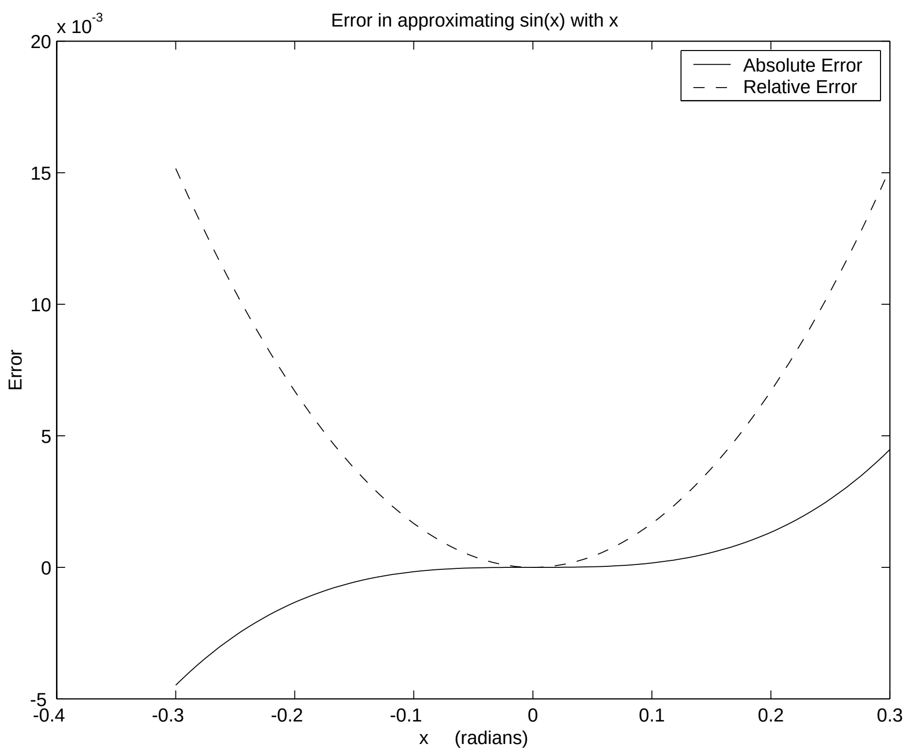
$$E_{\text{abs}} = x - \sin(x) = \frac{x^3}{3!} - \frac{x^5}{5!} + \ldots$$

And the relative error is

$$E_{\text{abs}} = \frac{x - \sin(x)}{\sin(x)} = \frac{x}{\sin(x)} - 1$$

Absolute and Relative Error (3)

Plot relative and absolute error in approximating $\sin(x)$ with x.



Although the absolute error is relatively flat around $x = 0$, the relative error grows more quickly. The relative error reflects the fact that the absolute value of $\sin(x)$ is small near $x = 0$.

Iteration termination (1)

An iteration generates a sequence of scalar values $x_k,\ k = 1, 2, 3, \ldots$. The sequence converges to a limit ξ if

$$|x_k - \xi| < \delta, \qquad \text{for all } k > N,$$

where δ is a small.

In practice, the test is expressed as

$$|x_{k+1} - x_k| < \delta, \quad \text{when } k > N.$$

Iteration termination (2)

Absolute convergence criterion

In words:

Iterate until $|x - x_{\text{old}}| < \Delta_a$

where Δ_a is the absolute convergence tolerance.

In MATLAB:

```
x = ...              %  initialize
xold = ...

while abs(x-xold) > deltaa

      xold = x;
      update x

end
```

Note: MATLAB does not have an "until" structure. The **while** construct involves a reverse in the direction of the inequality.

Iteration termination (3)

Relative convergence criterion

In words:

$$\text{Iterate until } \left| \frac{x - x_{\text{old}}}{x_{\text{old}}} \right| < \delta_r$$

where δ_r is the absolute convergence tolerance.

In MATLAB:

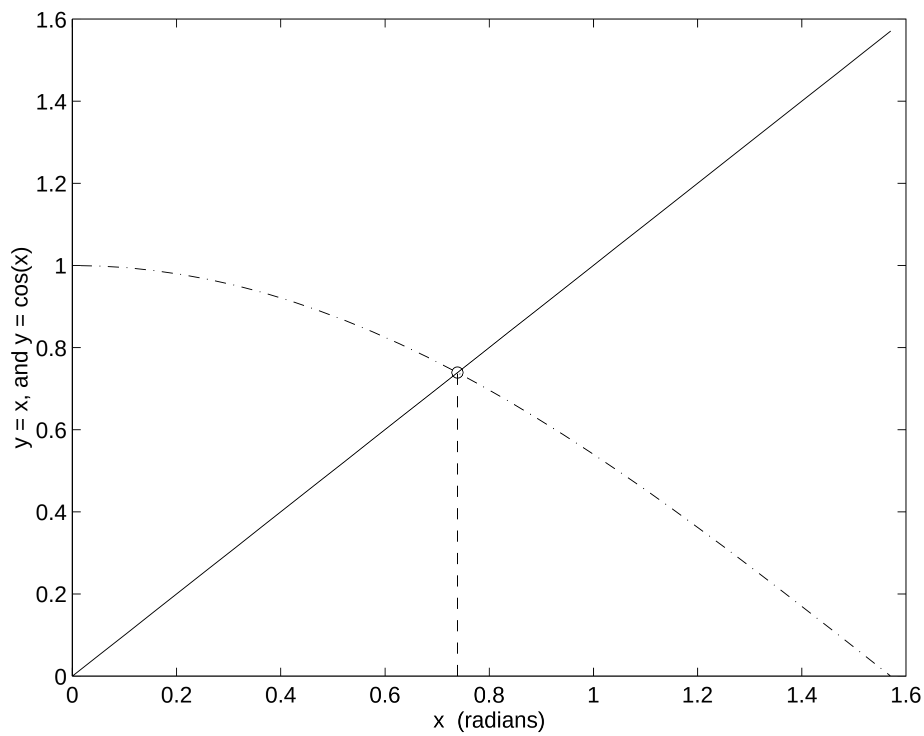
```
x = ...              %  initialize
xold = ...

while abs((x-xold)/xold) > deltar

      xold = x;
      update x

end
```

Example: Solve $\cos(x) = x$ (1)

Example: *Solve* $\cos(x) = x$ *with Fixed Point Iteration*
Obtain numerical solution to

$$\cos(x) = x$$

The solution lies at the intersection of $y = x$ and $y = \cos(x)$.



Example: Solve $\cos(x) = x$ (2)

In Chapter 6 we describe fixed point iteration as a method for obtaining a numerical approximation to the solution of a scalar equation. For now, trust that the follow algorithm will eventually give the solution.

1. Guess x_0
2. Set $x_{old} = x_0$
3. Update guess
$$x_{new} = \cos(x_{old})$$
4. If $x_{\text{new}} \approx x_{\text{old}}$ stop; otherwise set $x_{\text{old}} = x_{\text{new}}$ and return to step 3

Solve $\cos(x) = x$ (3)

MATLAB implementation

```
x0 = ...      %  initial guess
k = 0;
xnew = x0;
while NOT_CONVERGED & k < maxit
    xold = xnew;
    xnew = cos(xold);
    it = it + 1;
end
```

Solve $\cos(x) = x$ **(4)**

Bad test # 1

```
while xnew ~= xold
```

This test will be true unless `xnew` and `xold` are *exactly* equal. In other words, `xnew` and `xold` are equal only when their bit patterns are identical. This is bad because

- Test may never be met because of oscillatory bit patterns
- If test is eventually met, the iterations will probably do more work than needed

Solve $\cos(x) = x$ (5)

Bad test # 2

```
while (xnew-xold) > delta
```

Will always fail if `xnew` < `xold`

Solve $\cos(x) = x$ **(6)**

Workable test # 1: Absolute tolerance

```
while abs(xnew-xold) < delta
```

What value of `delta` to use?

Solve $\cos(x) = x$ **(7)**

Workable test # 2: Relative tolerance

```
while abs(xnew-xold)/xref > delta
```

The user supplies appropriate value of `xref`. For this particular iteration we could use `xref` = `xold`.

```
while abs(xnew-xold)/xold > delta
```

Note: For this particular problem the exact solution is $\mathcal{O}(1)$ so the absolute and relative convergence tolerance will terminate the calculations at roughly the same iteration.

Solve $\cos(x) = x$ (8)

Using the relative convergence tolerance, the code becomes

```
x0 = ...      %  initial guess
k = 0;
xnew = x0;
while (abs(xnew-xold)/xold > delta)  &  k < maxit
    xold = xnew;
    xnew = cos(xold);
    it = it + 1;
end
```

Note: Parentheses around `abs(xnew-xold)/xold > delta` are not needed, but are added to make the test clear.

Truncation Error

Consider the series for $\sin(x)$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

For small x, only a few terms are needed to get a good approximation to $\sin(x)$. The . . . terms are “truncated”

$$f_{\text{true}} = f_{\text{sum}} + \text{truncation error}$$

The size of the truncation error depends on x *and* the number of terms included in f_{sum}

Truncation of series for $\sin(x)$ (1)

```
function ssum = sinser(x,tol,n)
% sinser  Evaluate the series representation of the sine function
%
% Synopsis: ssum = sinser(x)
%           ssum = sinser(x,tol)
%           ssum = sinser(x,tol,n)
%
% Input:    x   = argument of the sine function, i.e., compute sin(x)
%           tol = (optional) tolerance on accumulated sum. Default:  tol = 5e-9
%                 Series is terminated when abs(T_k/S_k) < delta.   T_k is the
%                 kth term and S_k is the sum after the kth term is added.
%           n   = (optional) maximum number of terms. Default: n = 15
%
% Output:   ssum = value of series sum after nterms or tolerance is met

if nargin < 2,  tol = 5e-9;  end
if nargin < 3,  n = 15;      end

term = x;  ssum = term;         %  Initialize series
fprintf('Series approximation to sin(%f)\n\n  k      term         ssum\n',x);
fprintf('%3d  %11.3e  %12.8f\n',1,term,ssum);

for k=3:2:(2*n-1)
  term = -term * x*x/(k*(k-1));                  %  Next term in the series
  ssum = ssum + term;
  fprintf('%3d  %11.3e  %12.8f\n',k,term,ssum);
  if abs(term/ssum)<tol, break;  end             %  True at convergence
end
fprintf('\nTruncation error after %d terms is %g\n\n',(k+1)/2,abs(ssum-sin(x)));
```

Truncation of series for $\sin(x)$ (2)

For small x, the series for $\sin(x)$ converges in a few terms

```
>> s = sinser(pi/6);
Series approximation to sin(0.523599)

  k      term         ssum
  1   5.236e-001    0.52359878
  3  -2.392e-002    0.49967418
  5   3.280e-004    0.50000213
  7  -2.141e-006    0.49999999
  9   8.151e-009    0.50000000
 11  -2.032e-011    0.50000000

Truncation error after 6 terms is 3.56382e-014
```

The absolute truncation error in the series is small relative to the true value of $\sin(\pi/6)$

```
>> err = (s-sin(pi/6))/sin(pi/6)
err =
 -7.1276e-014
```

Truncation of series for $\sin(x)$ (3)

For larger x, the series for $\sin(x)$ converges more slowly

```
>> s = sinser(15*pi/6);
Series approximation to sin(7.853982)

  k      term         ssum
  1   7.854e+000    7.85398163
  3  -8.075e+001  -72.89153055
  5   2.490e+002  176.14792646
  7  -3.658e+002  -189.61411536
  9   3.134e+002  123.74757368
 11  -1.757e+002  -51.97719366
 13   6.948e+001   17.50733908
 15  -2.041e+001   -2.90292432
 17   4.629e+000    1.72578031
 19  -8.349e-001    0.89092132
 21   1.226e-001    1.01353632
 23  -1.495e-002    0.99858868
 25   1.537e-003    1.00012542
 27  -1.350e-004    0.99999038
 29   1.026e-005    1.00000064

Truncation error after 15 terms is 6.42624e-007
```

Increasing the number of terms will allow the series to converge within the default error tolerance of 5×10^{-9} used in `sinser`. A better solution to the slow convergence of the series are explored in Exercise 23.

Taylor Series

For a sufficiently continuous function $f(x)$ defined on the interval $x \in [a, b]$ we define the n^{th} order Taylor Series approximation $P_n(x)$

$$
\begin{aligned}
P_n(x) = & f(x_0) + (x - x_0) \left.\frac{df}{dx}\right|_{x=x_0} \\
& + \frac{(x - x_0)^2}{2} \left.\frac{d^2 f}{dx^2}\right|_{x=x_0} \\
& + \cdots + \frac{(x - x_0)^n}{n!} \left.\frac{d^n f}{dx^n}\right|_{x=x_0}
\end{aligned}
$$

Then there exists $\xi(x)$ with $x_0 \leq \xi(x) \leq x$ such that

$$
f(x) = P_n(x) + R_n(x)
$$

and

$$
R_n(x) = \frac{(x - x_0)^{(n+1)}}{(n+1)!} \left.\frac{d^{(n+1)} f}{dx^{(n+1)}}\right|_{x=\xi}
$$

Taylor Series (2)

Big "$\mathcal{O}$" notation

$$f(x) = P_n(x) + \mathcal{O}\left(\frac{(x - x_0)^{(n+1)}}{(n+1)!}\right)$$

or, for $x - x_0 = h$ we say

$$f(x) = P_n(x) + \mathcal{O}\left(h^{(n+1)}\right)$$

Taylor Series Example

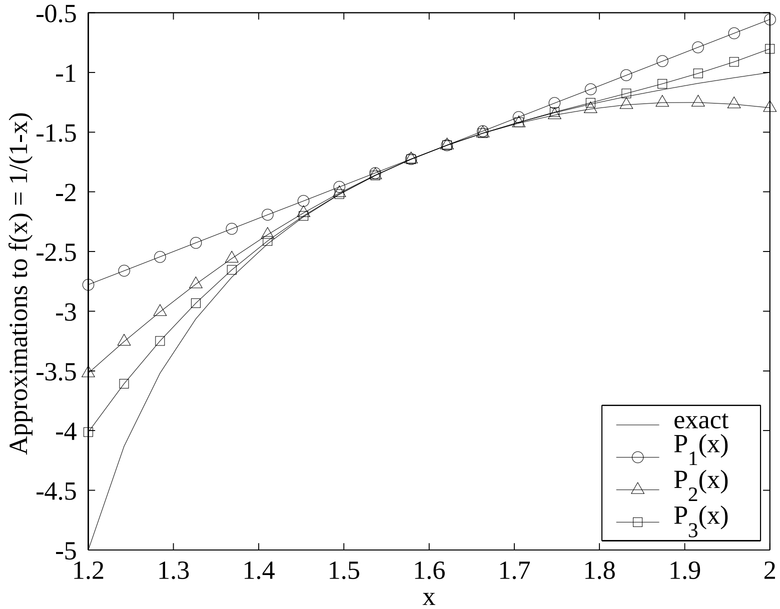
Consider the function

$$f(x) = \frac{1}{1-x}$$

The Taylor Series approximations to $f(x)$ of order 1, 2 and 3 are

$$\begin{aligned}
P_1(x) &= \frac{1}{1-x_0} \\
P_2(x) &= \frac{1}{1-x_0} + \frac{x-x_0}{(1-x_0)^2} \\
P_3(x) &= \frac{1}{1-x_0} + \frac{x-x_0}{(1-x_0)^2} + \frac{(x-x_0)^2}{(1-x_0)^3}
\end{aligned}$$

Taylor Series (4)



Roundoff and Truncation Errors (1)

Roundoff and truncation errors are both present in any numerical computation.

Example:

Finite difference approximation

A finite difference approximation to $f'(x) = df/dx$ is

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + \ldots$$

This approximation is said to be first order because the leading term in the truncation error is linear in h. Dropping the truncation error terms we obtain

$$f'_{fd}(x) = \frac{f(x+h) - f(x)}{h}$$

and

$$f'_{fd}(x) = f'(x) + \mathcal{O}(h)$$

Roundoff and Truncation Errors (2)

To study the roles of roundoff and truncation errors[1]., compute the finite difference approximation to $f'(x)$ when $f(x) = e^x$

$$f(x) = e^x \quad \Longrightarrow \quad f'(x) = e^x$$

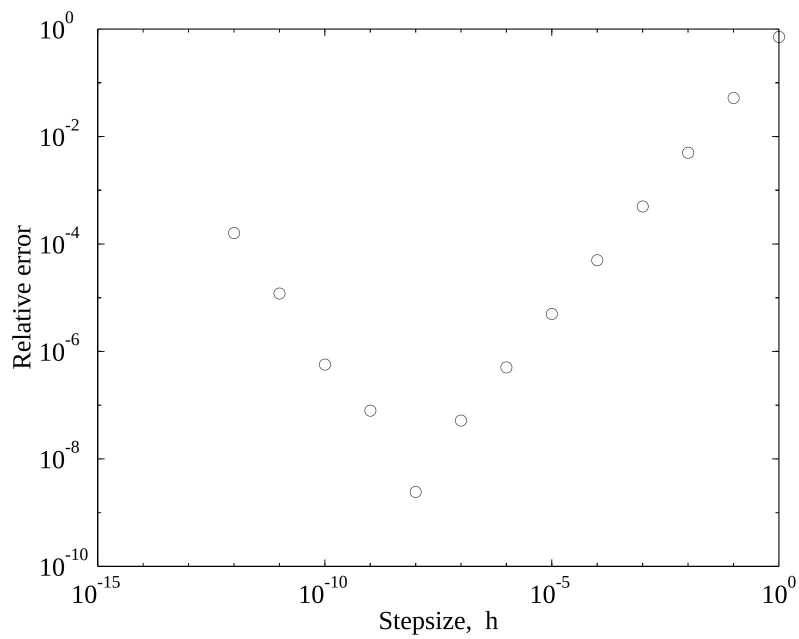
The relative error in the $f'_{fd}(x)$ approximation to $\dfrac{d}{dx}e^x$ is

$$E_{\mathrm{rel}} = \frac{f'_{fd}(x) - f'(x)}{f'(x)} = \frac{f'_{fd}(x) - e^x}{e^x}$$

[1]The finite difference approximation is usually applied in models of differential equations where $f(x)$ is *unknown*

Roundoff and Truncation Errors (3)

Evaluating E_{rel} for a range of h gives the following plot



Truncation error dominates at large h. Roundoff error in $f(x+h) - f(h)$ dominates as $h \rightarrow 0$.