
Chapter 1: Numerical Algorithms

Uri M. Ascher and Chen Greif
Department of Computer Science
The University of British Columbia
{ascher,greif}@cs.ubc.ca

Slides for the book

A First Course in Numerical Methods (published by SIAM, 2011)

<http://bookstore.siam.org/cs07/>

Goals of this chapter

- To explain what numerical algorithms are;
- to describe various sources and types of errors and how to measure them;
- to discuss algorithm properties and explain in a basic manner the notions of conditioning and stability;
- to illustrate the potentially damaging effect of roundoff errors.

Outline

- Scientific computing
- Numerical algorithms and errors
- Algorithm properties

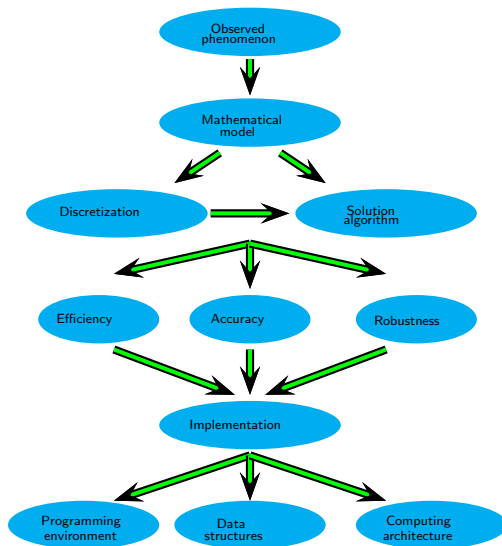


FIGURE: Scientific computing.

Methodology

- Approach the study of numerical algorithms by studying **errors**.
Not really glorious, but **useful**.
- Problem solving environment: **MATLAB** : numerical computing environment

Outline

- Scientific computing
- Numerical algorithms and errors
- Algorithm properties



How to measure errors


- * The most fundamental feature of numerical computing: inevitable presence of errors
- * The result of a computation is typically only approximate: want to ensure resulting error is tolerably small

- Can measure errors as **absolute** or **relative**, or a combination of both.
- The **absolute error** in v approximating u is $|u - v|$.
- The **relative error** (assuming $u \neq 0$) is $\frac{|u - v|}{|u|}$.

u	v	Absolute Error	Relative Error
1	0.99	0.01	0.01
1	1.01	0.01	0.01
-1.5	-1.2	0.3	0.2
100	99.99	0.01	0.0001
100	99	1	0.01

Source and type of errors

- Errors in the problem to be solved e.g. shape of objects
 - In the mathematical model (an approximation to reality) Simplifications
 - In input data Measurements: never infinitely accurate
- Approximation errors (in the numerical algorithm)
 - Discretization errors  arise from discretizations of continuous processes, such as differentiation, integration, interpolation, truncation
 - Convergence errors
 - Roundoff errors  arise in iterative methods (iterations cut prematurely)

 due finite precision representation of real numbers

Rounding errors occur if a real number (probably an intermediate result of some computation) is rounded to the next nearest machine number.

Example

Given smooth function $f(x)$, approximate derivative at some point $x = x_0$:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h},$$

for a small parameter value h .

Discretization: Function values $f(x)$ are available only at a discrete number of points, e.g. grid points $x_j = x_0 + kh$, k an integer

Discretization error:

$$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h} \right| \approx \frac{h}{2} |f''(x_0)|.$$

Results

Try for $f(x) = \sin(x)$ at $x_0 = 1.2$.

(So we are approximating $\cos(1.2) = 0.362357754476674... .$)

h	Absolute error
0.1	4.716676e-2
0.01	4.666196e-3
0.001	4.660799e-4
1.e-4	4.660256e-5
1.e-7	4.619326e-8

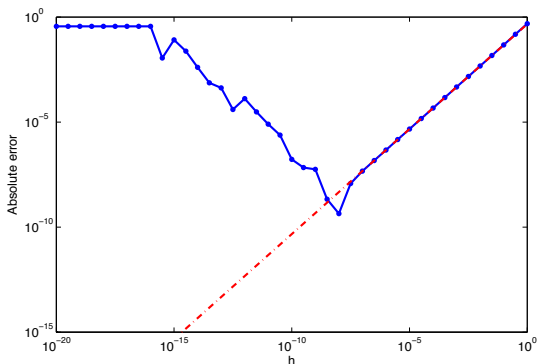
These results reflect the [discretization](#) error as expected.

Results for smaller h

h	Absolute error
1.e-8	4.361050e-10
1.e-9	5.594726e-8
1.e-10	1.669696e-7
1.e-11	7.938531e-6
1.e-13	6.851746e-4
1.e-15	8.173146e-2
1.e-16	3.623578e-1

These results reflect both [discretization](#) and [roundoff](#) errors.

F

Results for all h 

The solid curve interpolates the computed values of $|f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h}|$ for $f(x) = \sin(x)$, $x_0 = 1.2$. Shown in dash-dot style is a straight line depicting the discretization error without roundoff error.

Outline

- Scientific computing
- Numerical algorithms and errors
- Algorithm properties

Algorithm properties

Performance features that may be expected from a good numerical algorithm.

- ▶ **Accuracy**

Relates to errors. How accurate is the result going to be when a numerical algorithm is run with some particular input data.

- ▶ **Efficiency**

- ▶ How fast can we solve a certain problem?

Rate of convergence. Floating point operations (flops)

- ▶ How much memory space do we need?

- ▶ These issues may affect each other.

machine-independ
estimate of elem.
operations: +, -,
*, / ...

- ▶ **Robustness**

(Numerical) software should run under all circumstances.

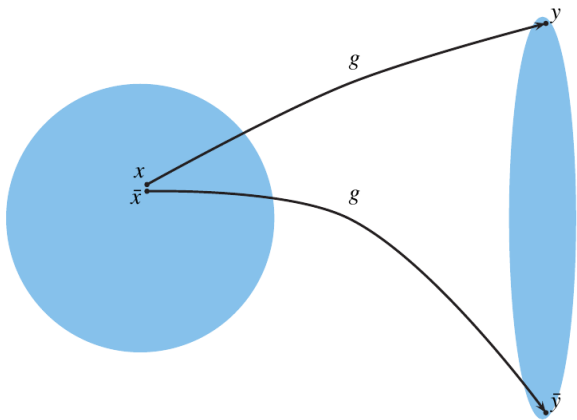
Should yield correct results to within an acceptable error or should fail gracefully if not successful.

Problem conditioning and algorithm stability

Qualitatively speaking:

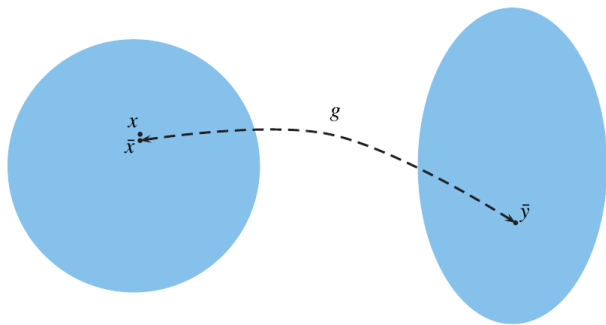
- The problem is **ill-conditioned** if a small perturbation in the data may produce a large difference in the result.
The problem is **well-conditioned** otherwise.
- The algorithm is **stable** if its output is the exact result of a slightly perturbed input.

Next, we'll see how bad **roundoff error accumulation** can be when an unstable algorithm is used.



Ill-conditioned problem of computing output values y from input values x by $y = g(x)$: when x is slightly perturbed to \bar{x} , the result $\bar{y} = g(\bar{x})$ is far from y .

A stable algorithm



An instance of a stable algorithm for computing $y = g(x)$: the output \bar{y} is the exact result, $\bar{y} = g(\bar{x})$, for a slightly perturbed input, i.e., \bar{x} which is close to the input x . Thus, if the algorithm is stable and the problem is well-conditioned, then the computed result \bar{y} is close to the exact y .

Unstable algorithm: an extreme example

Problem statement: evaluate the integrals

$$y_n = \int_0^1 \frac{x^n}{x+10} dx$$

for $n = 1, 2, \dots, 30$.

Algorithm development: observe that analytically

$$y_n + 10y_{n-1} = \int_0^1 \frac{x^n + 10x^{n-1}}{x+10} dx = \int_0^1 x^{n-1} dx = \frac{1}{n}.$$

Also

$$y_0 = \int_0^1 \frac{1}{x+10} dx = \ln(11) - \ln(10).$$

Algorithm:

- Evaluate $y_0 = \ln(11) - \ln(10)$.
- For $n = 1, \dots, 30$, evaluate $y_n = \frac{1}{n} - 10 y_{n-1}$.

Run program Example1.6.m

Roundoff error accumulation

- In general, if E_n is error after n elementary operations, cannot avoid linear roundoff error accumulation

$$E_n \simeq c_0 n E_0.$$

E_n : relative error
 c_0 : a constant

- Will not tolerate an **exponential** error growth such as

$$E_n \simeq c_1^n E_0 \quad \text{for some constant } c_1 > 1$$

– an **unstable algorithm**.

Roundoff errors

Since computer memory is finite practically no real number can be represented exactly in a computer.

- Roundoff error is generally inevitable in numerical algorithms involving real numbers.
- People often like to pretend they work with exact real numbers, ignoring roundoff errors, which may allow concentration on other algorithmic aspects.
- However, **carelessness may lead to disaster!**
- This chapter provides an overview of roundoff errors, including *floating point number representation, rounding error and arithmetic, the IEEE standard, and roundoff error accumulation.*

Rounding errors occur if a real number is rounded to the next nearest machine number.

The propagation of rounding errors from one floating point operation to the next is the most frequent source of numerical instabilities.

Goals of this chapter

Reading: Ascher & Greif: Chapter 2 and Chapter 1

- To describe how numbers are stored in a floating point system;
- to understand how standard floating point systems are designed and implemented;
- to get a feeling for the almost random nature of rounding error;
- to identify different sources of roundoff error growth and explain how to dampen their cumulative effect.

Outline

- Floating point systems
- The IEEE standard
- Roundoff error accumulation

We discuss floating point numbers as a representation of real numbers in the computer. A computer memory has a finite capacity

Real number representation: binary

The decimal system is convenient for humans; but computers prefer binary.

- In **binary** the (normalized) representation of a real number x is

$$\begin{aligned} x &= \pm(1.d_1d_2d_3\cdots d_{t-1}d_td_{t+1}\cdots) \times 2^e \\ &= \pm\left(1 + \frac{d_1}{2} + \frac{d_2}{4} + \frac{d_3}{8} + \cdots\right) \times 2^e, \end{aligned}$$

with binary digits $d_i = 0$ or 1 and exponent e .

- Floating point representation**: with a fixed number of digits t

$$\text{fl}(x) = \pm(1.\tilde{d}_1\tilde{d}_2\tilde{d}_3\cdots\tilde{d}_{t-1}\tilde{d}_t) \times 2^e$$

- How to determine digits \tilde{d}_i ?

A popular strategy is **Rounding**:

$$\text{fl}(x) = \begin{cases} \pm 1.d_1d_2d_3\cdots d_t \times 2^e & d_{t+1} = 0 \\ \text{to nearest even} & \text{otherwise} \end{cases}.$$

Alternatively, **Chopping** simply sets $\tilde{d}_i = d_i$, $i = 1, \dots, t$.

Real number representation: decimal

$$\frac{8}{3} \simeq \left(\frac{2}{10^0} + \frac{6}{10^1} + \frac{6}{10^2} + \frac{6}{10^3} \right) \times 10^0 = 2.666 \times 10^0.$$

An instance of the floating point representation

$$\begin{aligned} \text{fl}(x) &= \pm d_0.d_1 \cdots d_{t-1} \times 10^e \\ &= \pm \left(\frac{d_0}{10^0} + \frac{d_1}{10^1} + \cdots + \frac{d_{t-2}}{10^{t-2}} + \frac{d_{t-1}}{10^{t-1}} \right) \times 10^e \end{aligned}$$

for $t = 4$, $e = 0$.

Note that $d_0 > 0$: **normalized** floating point representation.

General floating point system

defined by (β, t, L, U) , where:

β : base of the number system (for binary, $\beta = 2$; for decimal, $\beta = 10$);

t : precision (number of digits);

L : lower bound on exponent e ;

U : upper bound on exponent e .

For each $x \in \mathbb{R}$ corresponds a normalized floating point representation

$$\text{fl}(x) = \pm \left(\frac{d_0}{\beta^0} + \frac{d_1}{\beta^1} + \cdots + \frac{d_{t-1}}{\beta^{t-1}} \right) \times \beta^e,$$

where $0 \leq d_i \leq \beta - 1$, $d_0 > 0$, and $L \leq e \leq U$.

Example

$$(\beta, t, L, U) = (10, 4, -2, 1).$$

Largest number is $9.999 \times 10^U = 99.99 \lesssim 10^{U+1} = 100$

Smallest positive number is $1.000 \times 10^L = 10^L = 0.01$

Total different fractions: $(\beta - 1) \times \beta^{t-1} = 9 \times 10 \times 10 \times 10 = 9,000$

Total different exponents: $U - L + 1 = 4$

Total different positive numbers: $4 \times 9,000 = 36,000$

Total different numbers in system: **72,001**

Error in floating point number representation

For the real number $x = \pm d_0.d_1d_2d_3 \cdots d_{t-1}d_t d_{t+1} \cdots \times \beta^e$,

- Chopping:

$$\text{fl}(x) = \pm d_0.d_1d_2d_3 \cdots d_{t-1} \times \beta^e$$

Then absolute error is clearly bounded by $\beta^{1-t} \cdot \beta^e$.

- Rounding:

$$\text{fl}(x) = \begin{cases} \pm d_0.d_1d_2d_3 \cdots d_{t-1} \times \beta^e & d_t < \beta/2 \\ \pm (d_0.d_1d_2d_3 \cdots d_{t-1} + \beta^{1-t}) \times \beta^e & d_t > \beta/2 \end{cases},$$

round to even in case of a tie.

Then absolute error is bounded by half the above, $\frac{1}{2} \cdot \beta^{1-t} \cdot \beta^e$.

So relative error is bounded by **rounding unit**

$$\eta = \frac{1}{2} \cdot \beta^{1-t}. \quad \begin{array}{l} \text{machine precision} \\ \text{machine epsilon} \end{array}$$

Floating point arithmetic

Important to use **exact rounding**: if x and y are machine numbers, then

$$\begin{aligned}\text{fl}(x \pm y) &= (x \pm y)(1 + \varepsilon_1), \\ \text{fl}(x \times y) &= (x \times y)(1 + \varepsilon_2), \quad |\varepsilon_i| \leq \eta. \\ \text{fl}(x/y) &= (x/y)(1 + \varepsilon_3),\end{aligned}$$

In other words: The result of a basic operation with two floating point numbers yields a result that is correct up to a relative error smaller than η .

Thus, the relative errors remain small after each such operation. This is achieved only using **guard digits** (intermediate higher precision). **Guard digits**: Extra digits used in interim calculations

Overflow, underflow, NaN

- **Overflow**: when $e > U$. (fatal)
- **Underflow**: when $e < L$. (non-fatal: set to 0 by default)
- **NaN**: Not-a-number. (e.g., $0/0$)

Remark on overflow

An **overflow** is obtained when a number is too large to fit into the floating point system in use.

Example: $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$

To make things simple, let $n = 2$. **i.e. a use a 2D vector**

If $x = (10^{60}, 1)^T$, then the exact result is $\|x\|_2 = 10^{60}$.

But in the course of the computation we have to form $x_1^2 = 10^{120}$.

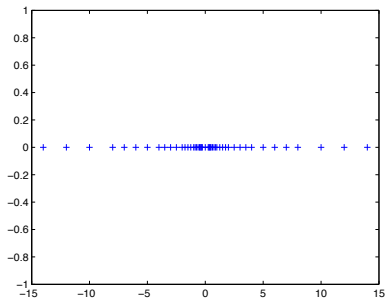
To avoid overflow, we **scale** with a positive scalar s , e.g.,

$s = \max(|x_i|)$

$$\|x\|_2 = s \sqrt{\left(\frac{x_1}{s}\right)^2 + \left(\frac{x_2}{s}\right)^2 + \cdots + \left(\frac{x_n}{s}\right)^2}$$

Spacing of floating point numbers

Run program [Example2_8Figure2_3](#)



Note the **uneven** distribution, both for large exponents and near **0**.

Outline

- Floating point systems
- The IEEE standard
- Roundoff error accumulation

IEEE standard

- Used by everyone today.
- **Binary**: use $\beta = 2$.
- **Exact rounding**: use guard digits to ensure that relative error in each elementary arithmetic operation is bounded by η .
- NaN
- Overflow and underflow
- **Subnormal numbers** near 0. (not covered in the course)

IEEE standard word

Use base $\beta = 2$.

Recall that with this base, in normalized numbers the first digit is always $d_0 = 1$ and thus it need not be stored.

Double precision (64 bit word)

$s = \pm$	$b = 11$ -bit exponent	$f = 52$ -bit fraction
-----------	------------------------	------------------------

Rounding unit:

$$\eta = \frac{1}{2} \cdot 2^{-52} \approx 1.1 \times 10^{-16}$$

Can have also single precision (32 bit word).

Then $t = 23$ and $\eta = 2^{-24} \approx 6.0 \times 10^{-8}$.

IEEE floating point numbers (cont.)

► **double**:

1 sign bit

11 bits exponent

52 bits mantissa

The value of a normalized 64-bit IEEE floating point number V is

$$V = (-1)^S \times 2^{(E-1023)} \times (1.M)$$

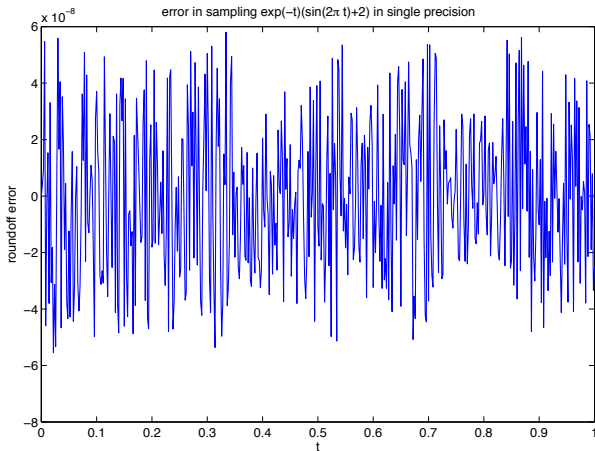
Normalized means, that $0 < E < 2047 = 2^{11} - 1$.

Outline

- Floating point systems
- The IEEE standard
- Roundoff error accumulation

The rough appearance of roundoff errors

Run program [Example2_2Figure2_2.m](#)



Note how the sign of the floating point representation error at nearby arguments t fluctuates as if randomly: as a function of t it is a “non-smooth” error.

Cancellation a special kind of rounding error

Suppose $z = x - y$, where $x \approx y$. Then

$$|z - \text{fl}(z)| \leq |x - \text{fl}(x)| + |y - \text{fl}(y)|,$$

from which it follows that the **relative** error satisfies

$$\frac{|z - \text{fl}(z)|}{|z|} \leq \frac{|x - \text{fl}(x)| + |y - \text{fl}(y)|}{|x - y|},$$

Numerator: OK.

Denominator is very close to zero if $x \approx y$. So the relative error in z could become large.

Illustration

Compute $y = \sqrt{x+1} - \sqrt{x}$ for $x = 100,000$ in a 5-digit decimal arithmetic.
(So $\beta = 10$, $t = 5$.)

- Naively computing $\sqrt{x+1} - \sqrt{x}$ results in the value 0.
- Instead use the identity

$$\frac{(\sqrt{x+1} - \sqrt{x})(\sqrt{x+1} + \sqrt{x})}{(\sqrt{x+1} + \sqrt{x})} = \frac{1}{\sqrt{x+1} + \sqrt{x}}.$$

- In 5-digit decimal arithmetic calculating the right hand side expression yields 1.5811×10^{-3} : correct in the given accuracy.

Example

Compute $y = \sinh(x) = \frac{1}{2}(e^x - e^{-x})$.

- Naively computing y at an x near 0 may result in a (meaningless) 0.
- Instead use Taylor's expansion due cancellation errors

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \dots$$

to obtain

$$\sinh(x) = x + \frac{x^3}{6} + \dots$$

- If x is near 0, can use $x + \frac{x^3}{6}$, or even just x , for an effective approximation to $\sinh(x)$. Now, round-off error is no longer an issue

So, a good library function would compute $\sinh(x)$ by the regular formula (using exponentials) for $|x|$ not very small, and by taking a term or two of the Taylor expansion for $|x|$ very small.

Subtracting two almost equal numbers

Let $a = 1.2$ and $b = 1.1$. We compare two algorithms to compute $a^2 - b^2 = (a + b)(a - b)$ in decimal arithmetic with two essential digits.

$$a^2 = 1.44 \implies \text{fl}(a^2) = 1.4$$

$$b^2 = 1.21 \implies \text{fl}(b^2) = 1.2$$

$$\text{fl}(a^2) - \text{fl}(b^2) = 0.2 \quad (13\% \text{ error})$$

$$a + b = \text{fl}(a + b) = 2.3$$

$$a - b = \text{fl}(a - b) = 0.10$$

$$(a + b)(a - b) = 0.23 \quad (0\% \text{ error})$$