2. Computer Arithmetic

- floating–point numbers
- floating–point representation
- floating–point error Analysis
- sources of error in numerical computation
- stable & unstable Computations
- conditioning of a problem

Floating–point numbers

computer users read numbers in **decimal system**

$$9.75 = 9 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2}$$

computer internally work with **binary system**

$$(1001.11)_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

example: change $1/10$ to the binary system

$$\frac{1}{10} = (0.0001\ 1001\ 1001\ 1001\ 1001 \ldots)_2$$

Rounding up and Truncation

x is a positive decimal number with m digits to the right of the decimal point.

Rounding: round x up to n decimal places

- if $(n+1)$st digit is $0, 1, 2, 3$, or 4 then the nth digit is not changed
- if $(n+1)$st digit is $5, 6, 7, 8$, or 9 then the nth digit is increased by one
- the remaining digits are discarded.

examples: seven-digit numbers are rounded to four digits

$$\begin{array}{ccc} 0.1735499 & \rightarrow & 0.1735 \\ 0.9999500 & \rightarrow & 1.0000 \\ 0.4321609 & \rightarrow & 0.4322 \end{array}$$

Fact: if $\tilde{x}$ is the rounded-up n–digit approximation to x, then

$$|x - \tilde{x}| \leq \frac{1}{2} \times 10^{-n}$$

Truncation: truncate a number to n digits is to *discard* all digits beyond the nth digit.

examples: seven-digit numbers are truncated to four digits

$$\begin{aligned} 0.1735499 &\rightarrow 0.1735 \\ 0.9999500 &\rightarrow 0.9999 \\ 0.4321609 &\rightarrow 0.4321 \end{aligned}$$

Fact: if $\hat{x}$ is the truncated (chopped) n–digit approximation of x then

$$|x - \hat{x}| \leq 10^{-n}$$

Normalized scientific notation

normalized scientific notation in **decimal system**:

- shift decimal point with appropriate powers of 10
- all digits are to the right of the decimal point and the first digit displayed is not 0

example: $732.5051 = 0.7325051 \times 10^3$

a nonzero real number x can be represented in form

$$x = \pm r \times 10^n$$

where $\frac{1}{10} \leq r < 1$ and n is an integer normalized scientific notation in **binary system**:

$$x = \pm q \times 2^m$$

where $\frac{1}{2} \leq q < 1$,

- q is called the **mantissa**,
- m is an integer and called the **exponent**

another version: leading binary digit 1 appears to the left of the binary point

$$x = \pm q \times 2^m, \quad q = (1.f)_2$$

where $1 \leq q < 2$.

Floating–point representation

floating–point representation for a single-precision real number

$$x = \pm q \times 2^m$$

in a **32-bit computer** is divided into 3 fields.

- sign of real number x (s) 1 bit
- biased exponent (integer e) 8 bits
- mantissa part (real number f) 23 bits

sign of mantissa s	biased exponent e	normalized mantissa f

values of bit strings are decoded as normalized floating–point form

$$x = (-1)^s q \times 2^m, \quad q = (1.f)_2, \quad m = e - 127$$

note: the most significant digit in q is 1 and is not stored

Fact:

$$1 \leq q < 2, \quad 0 < e < 255, \quad -126 \leq m \leq 127$$

$e = 0$ and $e = 255$ are reserved for special cases such as $\pm 0, \pm\infty$ and NaN

32 bit computer can handle numbers

- smallest number $2^{-126} \approx 1.2 \times 10^{-38}$
- largest number $(2 - 2^{-23})2^{127} \approx 3.4 \times 10^{38}$

double precision or extended precision uses two computer words and slows down calculation

Machine rounding

- round to nearest: the closer of two machine numbers of the real number is selected
- round to even: in case of halfway between two machine numbers, even machine number is chosen
- directed rounding such as round toward 0 (truncation)

Nearby machine numbers

what is the machine number closest to x?

$$x = (1.a_1a_2\dots a_{22}a_{23}a_{24}a_{25}\dots)_2 \times 2^m$$

- chopping

$$x_- = (1.a_1a_2\dots a_{22}a_{23})_2 \times 2^m$$

- rounding up

$$x_+ = \left((1.a_1a_2\dots a_{22}a_{23})_2 + 2^{-23}\right) \times 2^m$$

x can be represented better either by x_- or x_+ (depending on the value of x)

$$x = (1.0011\cdots 0101)_2 \times 2^3$$

$$x_- = (1.0011\cdots 01)_2 \times 2^3 \implies |x - x_-| = (1 \cdot 2^{-24}) \times 2^3$$

$$x_+ = (1.0011\cdots 10)_2 \times 2^3 \implies |x_+ - x| = [1 \cdot 2^{-22} - (1 \cdot 2^{-23} + 1 \cdot 2^{-24})] \times 2^3$$

first case: x is represented better by x_-

$$|x - x_-| \leq \frac{1}{2}|x_+ - x_-| = \frac{1}{2} \times 2^{m-23} = 2^{m-24}$$

relative error is bounded by

$$\frac{|x - x_-|}{x} \leq \frac{2^{m-24}}{q \times 2^m} = \frac{1}{q} \times 2^{-24} \leq 2^{-24}$$

second case: x is closer to x_+ than to x_-

$$|x - x_+| \leq \frac{1}{2}|x_+ - x_-| = 2^{m-24}$$

relative error is bounded by

$$\frac{|x - x_+|}{x} \leq 2^{-24}$$

Overflow vs Underflow

in a 32-bit computation, a number is produced of the form

$$\pm q \times 2^m$$

we call a computation is **overflow** if m is outside the range permitted

$$m > 127$$

and we call a computaton is **underflow** if m is too small,

$$m < -126$$

- IEEE standard uses a kind of *extended floating point* system to allow for results `Inf` and `NaN`
- it includes rules such as `x/Inf` gives 0 or `x/0` yields $\pm$`Inf`

Roundoff error

let x^* be the machine number closest to x and $\delta = (x^* - x)/x$

$$\frac{|x - x^*|}{x} \leq 2^{-24}$$

$\mathbf{fl}(x)$ is used to denote x^*, that is,

$$\mathbf{fl}(x) = x^* = x(1 + \delta), \quad |\delta| \leq 2^{-24}$$

thus, 2^{-24} is the **unit roundoff error** for 32-bit computers

Fact: number of bits of mantissa directly relates to unit roundoff error and determines accuracy of calculation for computer with base β and mantissa n places

$$\mathbf{fl}(x) = x(1+\delta), \quad |\delta| \leq \epsilon$$

where

- $\epsilon = (1/2)\beta^{1-n}$ if we implement correct rounding
- $\epsilon = \beta^{1-n}$ if we implement chopping

ϵ is the unit roundoff error and is a charateristic of a computing machine

example: find the nearest machine number of $x = 2/3$

to find the binary representation of $2/3$, we write

$$x = 2/3 = (0.a_1a_2a_3\cdots)_2$$

to find a_1, multiply x by 2

$$2x = 4/3 = (a_1.a_2a_3a_4\cdots)_2 \quad \Longrightarrow \quad a_1 = 1 \quad (\because 4/3 > 1)$$

substracting 1 from both sides

$$1/3 = (0.a_2a_3a_4\cdots)_2$$

multiplying 2 on both sides

$$2/3 = (a_2.a_3a_4a_5\cdots)_2 \quad \Longrightarrow \quad a_2 = 0 \quad (\because 2/3 < 1)$$

repeat the previous steps, we obtain

$$x = 2/3 = (0.1010\cdots)_2 = (1.010101\cdots)_2 \times 2^{-1}$$

the two nearby machine numbers are

$$x_- = (1.010101\cdots010)_2 \times 2^{-1}, \quad x_+ = (1.010101\cdots011)_2 \times 2^{-1}$$

and the absolute errors are

$$\begin{aligned} x - x_- &= (0.1010\cdots)_2 \times 2^{-24} = 2/3 \times 2^{-24} \\ x_+ - x &= (x_+ - x_-) - (x - x_-) = 2^{-24} - 2/3 \times 2^{-24} = 1/3 \times 2^{-24} \end{aligned}$$

hence, we set $\mathbf{fl}(x) = x_+$ (the nearest machine number)

the relative roundoff error is

$$\frac{|\mathbf{fl}(x) - x|}{|x|} = \frac{1/3 \times 2^{-24}}{2/3} = 2^{-25}$$

Floating–point error analysis

for any real number x within the range of the 32-bit computer

$$\mathbf{fl}(x) = x(1+\delta), \quad |\delta| \leq 2^{-24}$$

if x, y are machine numbers, we have

$$\mathbf{fl}(x \odot y) = (x \odot y)(1+\delta) \quad |\delta| \leq 2^{-24}$$

where $\odot$ is one of the four arithmatic operations; $+ - \times \div$

roundoff error must be expected in every arithmatic operation !

example: both 2^{-1} and 2^{-25} are machine numbers, but so is $2^{-1} + 2^{-25}$?

if x, y are machine numbers and assume arithmetic operations satisfy

$$\mathbf{fl}(x \odot y) = (x \odot y)(1+\delta), \quad |\delta| \leq \epsilon$$

we often compute the roundoff error from a series of arithmetic operations

for example,

$$\begin{aligned}
\mathbf{fl}[x(y+z)] &= [x\mathbf{fl}(y+z)](1+\delta_1), \quad |\delta_1| \leq 2^{-24} \\
&= [x(y+z)(1+\delta_2)](1+\delta_1), \quad |\delta_2| \leq 2^{-24} \\
&= x(y+z)(1+\delta_1+\delta_2+\delta_1\delta_2) \\
&\approx x(y+z)(1+\delta_1+\delta_2) \\
&= x(y+z)(1+\delta_3), \quad |\delta_3| \leq 2^{-23}
\end{aligned}$$

if x, y are *not* machine numbers, one should expect

$$\mathbf{fl}(\mathbf{fl}(x) \odot \mathbf{fl}(y)) = (x(1+\delta_1) \odot y(1+\delta_2))(1+\delta_3) \quad |\delta_i| \leq 2^{-24}$$

Relative error in adding

given a machine with a unit roundoff error ϵ and that

$$x_0, x_1, \ldots, x_n \quad \text{are positive machine numbers}$$

then the relative roundoff error in computing the sum of $n+1$ numbers,

$$x_0 + x_1 + x_2 + \cdots + x_n$$

is at most $(1+\epsilon)^n - 1$ and should not exceed $n\epsilon$

Proof. define S_k (actual sum)

$$S_k = x_0 + x_1 + \ldots + x_k$$

the computer calculates S_k^* (computed sum) recursive formula for S_k

$$S_0 = x_0, \quad S_{k+1} = S_k + x_{k+1}$$

recursive formula for S_k^*

$$S_0^* = x_0, \quad S_{k+1}^* = \mathbf{fl}(S_k^* + x_{k+1})$$

let $|\rho_k|$ be relative error between actual S_k and computed sum S_k^*

$$\rho_k = \frac{S_k^* - S_k}{S_k} \quad \text{(relative error after } k \text{ steps)}$$

let $|\delta_k|$ be relative error in computing $S_k^* + x_{k+1}$

$$\delta_k = \frac{S_{k+1}^* - (S_k^* + x_{k+1})}{S_k^* + x_{k+1}} \quad \text{(relative error at the } (k+1)\text{th step)}$$

it can be shown that

$$
\begin{aligned}
\rho_{k+1} &= \frac{S_{k+1}^* - S_{k+1}}{S_{k+1}} \\
&= \frac{(S_k^* + x_{k+1})(1+\delta_k) - S_{k+1}}{S_{k+1}} \\
&= \frac{(S_k(1+\rho_k) + x_{k+1})(1+\delta_k) - S_{k+1}}{S_{k+1}} \\
&= \frac{(S_{k+1} + S_k\rho_k)(1+\delta_k) - S_{k+1}}{S_{k+1}} \\
&= \frac{S_{k+1}\delta_k + S_k\rho_k(1+\delta_k)}{S_{k+1}} \\
&= \delta_k + \rho_k(S_k/S_{k+1})(1+\delta_k)
\end{aligned}
$$

at each iteration, δ_k is directly added up to ρ_{k+1}

since $S_k/S_{k+1} < 1$ and $|\delta_k| \leq \epsilon$, we conclude

$$|\rho_{k+1}| \leq \epsilon + |\rho_k|(1+\epsilon)$$

or equivalently,

$$|\rho_{k+1}| \leq \epsilon + |\rho_k|\theta, \qquad \theta = 1+\epsilon$$

by successive inequalities and $|\rho_0| = 0$, we have

$$|\rho_n| \;\leq\; \epsilon + \theta\epsilon + \theta^2\epsilon + \cdots + \theta^{n-1}\epsilon \;=\; \epsilon\frac{(\theta^n - 1)}{(\theta - 1)} \;= (1+\epsilon)^n - 1$$

by the Binomial theorem, we have

$$(1+\epsilon)^n = 1 + \begin{pmatrix} n \\ 1 \end{pmatrix}\epsilon + \begin{pmatrix} n \\ 2 \end{pmatrix}\epsilon^2 + \cdots + \epsilon^n$$

by neglecting the higher order term in ϵ^k, $k \geq 2$

$$|\rho_n| \leq (1+\epsilon)^n - 1 \approx n\epsilon$$

Absolute and Relative Errors

let x^* be approximated number of x

absolute error

$$|x - x^*|$$

relative error

$$\frac{|x - x^*|}{|x|}$$

- absolute error needs a knowledge about the magnitude of x
- relative error is often more significant and useful

Machine epsilon

the machine epsilon u is the largest point number x such that

$$1 + x = 1$$

$i.e.$, $x + 1$ cannot be distinguished from 1 on the computer:

$$u = \max \{x \mid 1 + x = 1, \quad \text{in computer arithmetic}\}$$

example: a three digit decimal computer that uses rounding

$$x_1 = 1.00 \times 10^{-2} \Longrightarrow 1 + x_1 = 1.00 + 0.01 = 1.01 \neq 1.00$$

$$x_2 = 1.00 \times 10^{-3} \Longrightarrow 1 + x_2 = 1.00 + 0.001 = 1.001 \to 1.00 = 1.00$$

$$x_3 = 5.00 \times 10^{-3} \Longrightarrow 1 + x_3 = 1.00 + 0.005 = 1.005 \to 1.01 \neq 1.00$$

x_1 is too large, x_2 is too small, x_3 is a bit large

$$u = 4.99 \times 10^{-3} \Longrightarrow 1 + u = 1.00 + 0.00499 = 1.00499 \to 1.00 = 1.00$$

Loss of significance

numerical analysis is to understand and control various kinds of errors

- roundoff error
- loss of significance or precision, *e.g.*, subtraction of nearly equal quantities, evaluation of functions,

a remedy to loss of significance is to carefully write program

example: the assignment statement

$$y = \sqrt{x^2 + 1} - 1$$

involves substractive calculation and loss of significance for small values of x. to avoid the difficulty, this can be rewritten as

$$y = \frac{x^2}{\sqrt{x^2 + 1} + 1}$$

Evaluation of Functions

Evaluating some $f(x)$ for very large x can cause a drastic loss of significant digits

consider cosine function which has periodicity property

$$\cos(x + 2n\pi) = \cos(x), \quad n = \mathbf{Z}$$

and other properties

$$\cos(-x) = \cos(x) = -\cos(\pi - x)$$

example:

$$\cos(33278.21) = \cos(33278.21 - 5296\pi) = \cos(2.46)$$

there is a library subroutine called range reduction which exploits these properties

Theorem on loss of precision

if x and y are positive normalized floating-point binary machine numbers s.t.

$$x > y \qquad \text{and} \qquad 2^{-q} \leq 1 - (y/x) \leq 2^{-p}$$

then at most q and at least p significant bits are lost in $x - y$

Proof. suppose x and y are in the normalized form

$$x = r \times 2^n, \qquad y = s \times 2^m$$

to prove the upperbound, shift the exponent of y such that

$$x - y = (r - s \times 2^{m-n}) \times 2^n$$

the mantissa of this number satisfies

$$r - s \times 2^{m-n} = r\left(1 - \frac{s \times 2^m}{r \times 2^n}\right) = r\left(1 - \frac{y}{x}\right) < 2^{-p}$$

to normalize the mantissa, at least a p-bit shift to the left is required

to prove the lower bound, shift the exponent of x such that

$$x - y = ((r \times 2^{n-m} - s) \times 2^m)$$

the mantissa of this number satisfies

$$r \times 2^{n-m} - s = s\left(\frac{r \times 2^n}{s \times 2^m} - 1\right) = s\left(\frac{x}{y} - 1\right) \geq 2^{-q}$$

to normalize the mantissa, at most a q-bit shift to the left is required for example, the mantissa of $x - y$ satisfies

$$2^{-4} \leq (0.00011100 \cdots 101)_2 \leq 2^{-3}$$

the normalized form of the mantissa is

$$(1.1100 \cdots 1010000)_2$$

four *spurious* zeros (not significant bits) were added to the right end

Sources of error in numerical computation

example: evaluate a function $f : \mathbf{R} \to \mathbf{R}$ at a given x ($e.g.$, $f(x) = \sin x$)

sources of error in the result:

- x is not exactly known
  - measurement errors
  - errors in previous computations

  $\longrightarrow$ how sensitive is $f(x)$ to errors in x?

- the algorithm for computing $f(x)$ is not exact
  - discretization ($e.g.$, the algorithm uses a table to look up $f(x)$)
  - truncation ($e.g.$, f is computed by truncating a Taylor series)
  - rounding error during the computation

  $\longrightarrow$ how large is the error introduced by the algorithm?

The condition of a problem

sensitivity of the solution with respect to errors in the data

- a problem is **well-conditioned** if small errors in the data produce small errors in the result

- a problem is **ill-conditioned** if small errors in the data may produce large errors in the result

rigorous definition depends on what 'large error' means (absolute or relative error, which norm is used, . . .)

example: function evaluation

$$y = f(x), \qquad y + \Delta y = f(x + \Delta x)$$

- absolute error

$$|\Delta y| \approx |f'(x)||\Delta x|$$

  ill-conditioned with respect to absolute error if $|f'(x)|$ is very large

- relative error

$$\frac{|\Delta y|}{|y|} \approx \frac{|f'(x)||x|}{|f(x)|}\frac{|\Delta x|}{|x|}$$

  ill-conditioned w.r.t relative error if $|f'(x)||x|/|f(x)|$ is very large

the factor $|xf'(x)|/|f(x)|$ serves as a **condition number** for the problem

example: $f(x) = \arcsin x$

a straightforward calculation shows that

$$\frac{xf'(x)}{f(x)} = \frac{x}{\sqrt{1-x^2}\arcsin x}$$

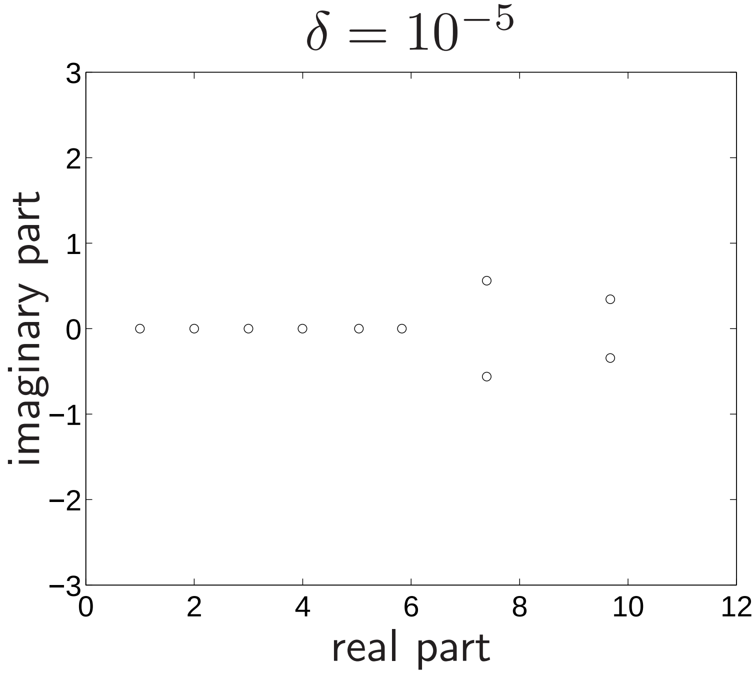
hence, for x near 1, the condition number becomes infinite

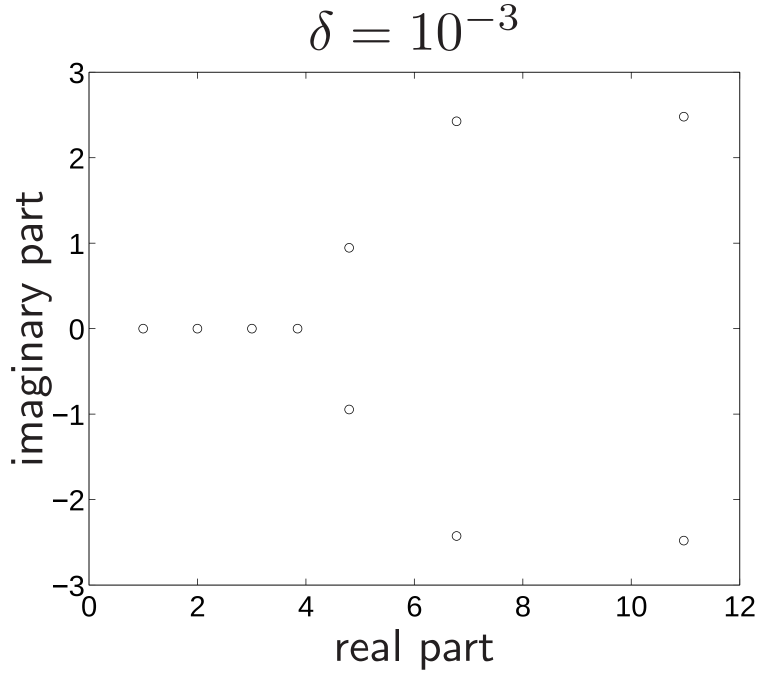
small relative errors in x may lead to large relative errors in $\arcsin x$ near $x = 1$

Roots of a polynomial

$$p(x) = (x-1)(x-2)\cdots(x-10) + \delta \cdot x^{10}$$

roots of p computed by Matlab for two values of δ





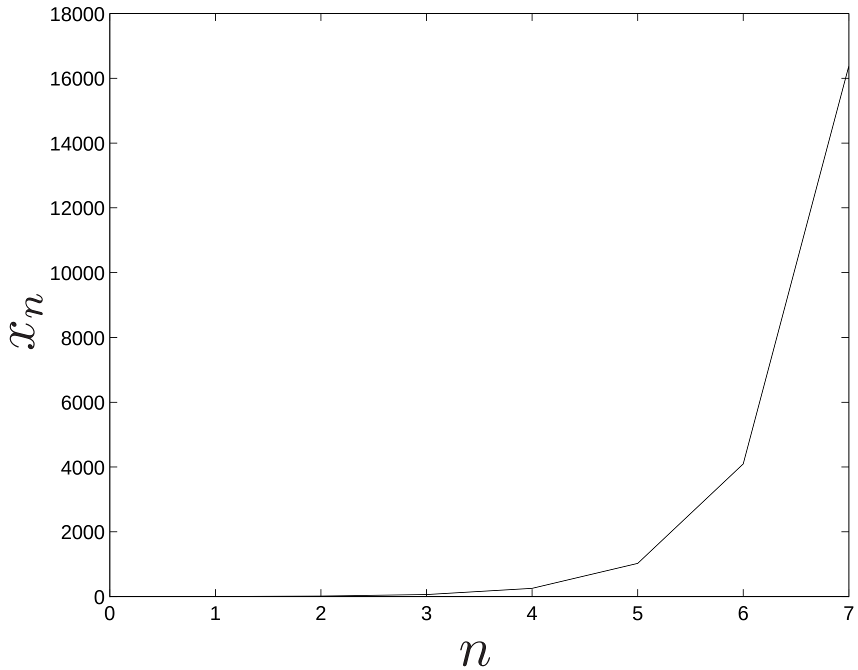
roots are very sensitive to errors in the coefficients

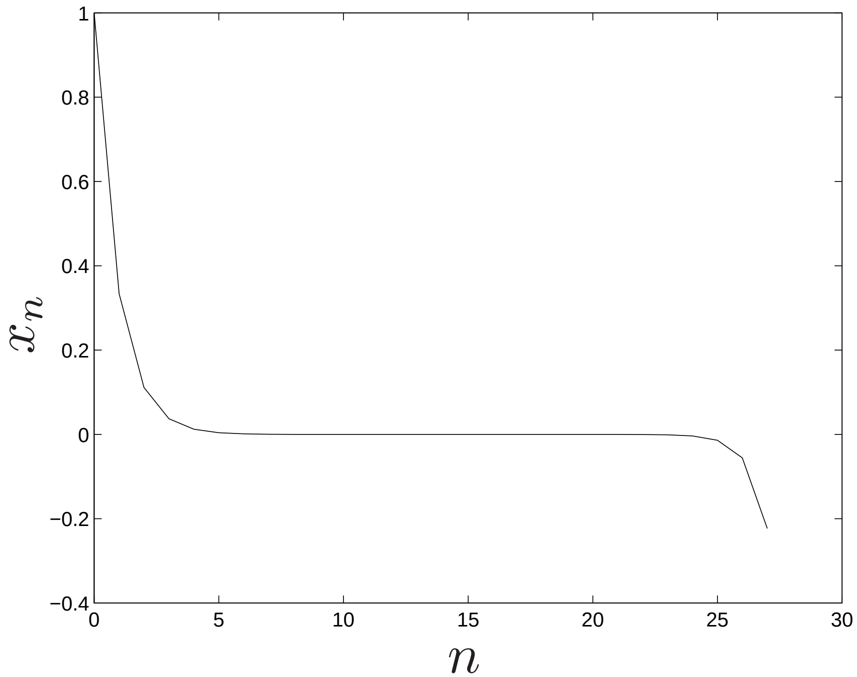
Stable & Unstable computations

example 1: $x_{n+1} = (13/3)x_n - (4/3)x_{n-1}$

case 1: $x_0 = 1, x_1 = 4$ and the exact solution is $x_n = 4^n$

case 2: $x_0 = 1, x_1 = 1/3$ and the exact solution is $x_n = (1/3)^n$





(Left.) case 1: accurate *(Right.)* case 2: inaccurate

a numerical process is **stable** when

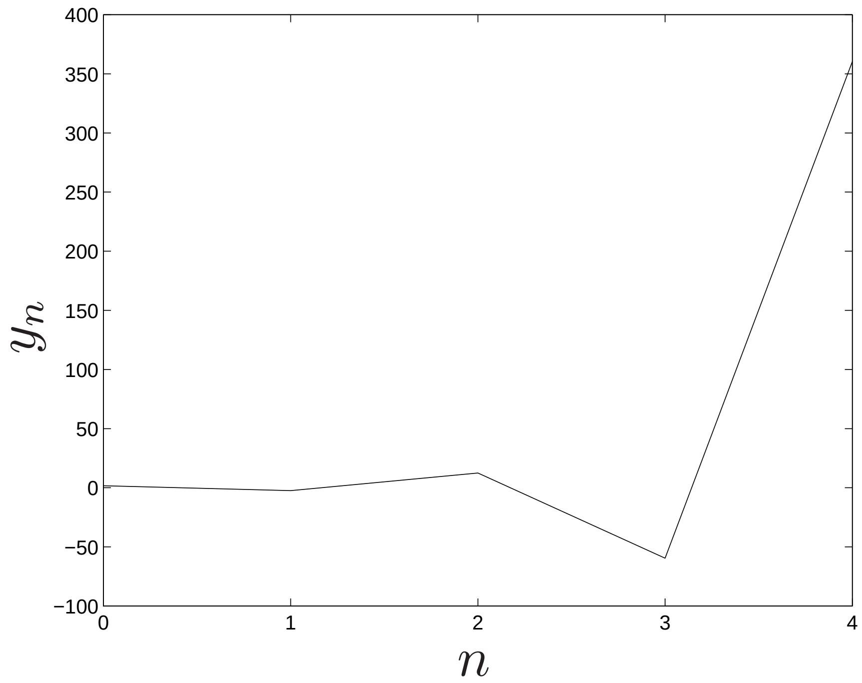
- small absolute errors made at one stage are magnified in subsequent stages
- but the relative errors are NOT seriously degraded

a numerical process is **unstable** when

- small absolute errors made at one stage are magnified in subsequent stages
- and the relative errors are *seriously degraded*

example 2: $y_n = \int_0^1 x^n e^x \, dx$

we apply integration by parts to the integral defining y_{n+1}, thus



$$y_{n+1} = e - (n+1)y_n, \quad y_0 = e - 1$$

errors influence the correct values of y_n

thus, the numerical solution is wrong

the correct solution is that y_n tends to zero as $n \to \infty$

$$\lim_{n\to\infty} y_n = 0, \quad \lim_{n\to\infty} (n+1)y_n = e$$

Summary

the **conditioning** of a mathematical problem

- sensitivity of the solution with respect to perturbations in the data
- ill-conditioned problems are 'almost unsolvable' in practice ($i.e.$, in the presence of data uncertainty): even if we solve the problem exactly, the solution may be meaningless
- a property of a problem, independent of the solution method

stability of an algorithm

- accuracy of the result in the presence of rounding error
- a property of a numerical algorithm

precision of a computer

- a machine property (usually IEEE double precision, $i.e.$, about 15 significant decimal digits)
- a bound on the *rounding error* introduced when representing numbers in finite precision

accuracy of a numerical result

- determined by: machine precision, accuracy of the data, stability of the algorithm, . . .
- usually much smaller than 16 significant digits

References

Chapter 2 in

D. Kincaid and W. Cheney, *Numerical Analysis: Mathematics of Scientific Computing*, 3rd edition, Brooks/Cole, 2002.

Section 1.3 in

J. F. Epperson, *An Introduction to Numerical Methods and Analysis*, Johy Wiley & Sons, Inc., 2007.

Lecture notes on

Problem condition and numerical stability, EE103, L. Vandenberhge, UCLA