SCK•CEN
STUDIECENTRUM VOOR KERNENERGIE
CENTRE D'ETUDE DE L'ENERGIE NUCLEAIRE

SCK•CEN ACADEMY
FOR NUCLEAR SCIENCE AND TECHNOLOGY

Can you count on your computer?[1]

Gert Van den Eynde

SCK•CEN

September 2nd, 2016

[1] Thanks to N. Higham for this wonderful pun-title

Let’s play: The Golden Ratio

Fibonacci

A simple program that

- calculates the first **N** Fibonacci numbers,
- calculates the ratio $\frac{f_i}{f_{i-1}}$ for $i = 1, 2, \ldots$
- plots this ratio as a function of the index

Let’s play: The Golden Ratio

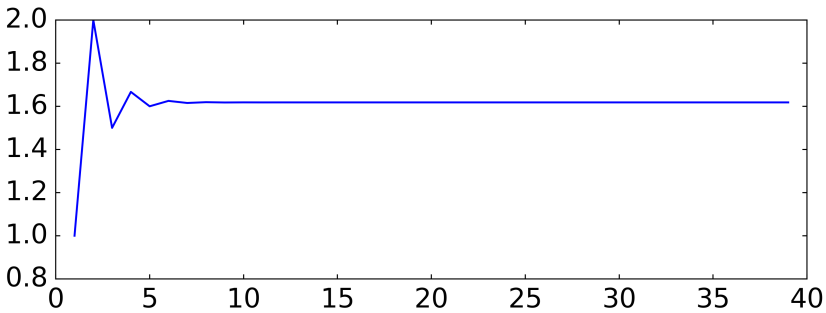
Fibonacci

```
>>> import numpy as np
>>> import matplotlib as mpl
>>> from matplotlib import pyplot as plt

>>> def fib(n):
...     f = np.zeros(n,dtype=np.int32)
...     g = np.zeros(n-1)
...     f[0] = 1
...     f[1] = 1
...     g[0] = f[1]/f[0]
...     for i in range(2,n):
...         f[i] = f[i-1]+f[i-2]
...         g[i-1] = f[i]/f[i-1]
...     return f,g
...
>>> N = [40, 50]
>>> for n in N:
...     f,g = fib(n)
...     plt.figure(figsize=(12,4))
...     plt.style.use('seaborn-notebook')
...     plt.plot(np.linspace(1,n-1,n-1),g)
...     plt.tick_params(labelsize=24)
...     plt.savefig('Fibonacci_{0}.png'.format(n),dpi=300,bbox_inches = 'tight')
...
<matplotlib.figure.Figure object at 0x0000000004993D30>
[<matplotlib.lines.Line2D object at 0x00000000065B3748>]
<matplotlib.figure.Figure object at 0x00000000067326D8>
[<matplotlib.lines.Line2D object at 0x0000000006FA4DA0>]
```

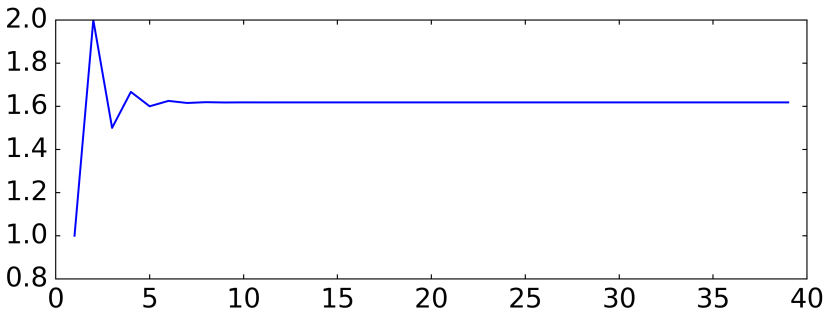
Let’s play: The Golden Ratio

$N = 40$

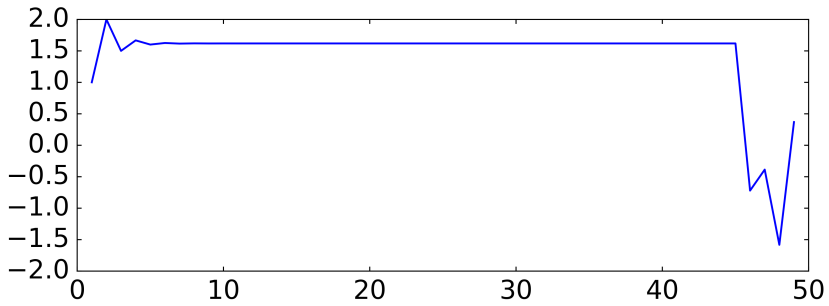


Let’s play: The Golden Ratio

$N = 40$

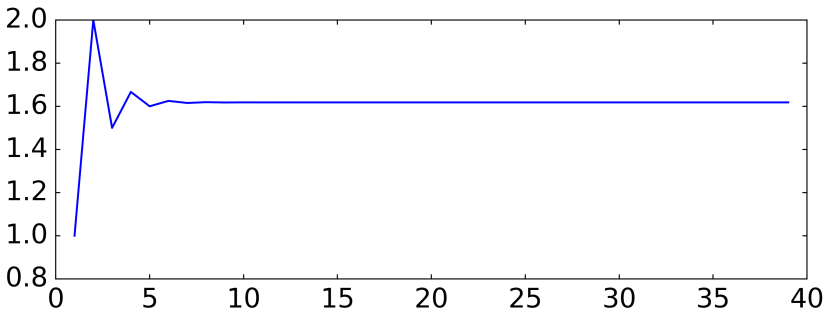


$N = 50$

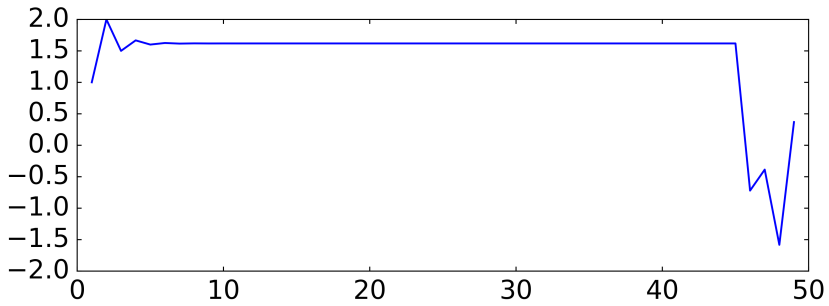


Let's play: The Golden Ratio

$N = 40$



$N = 50$



What is happening?

Introduction

Representing floating point numbers

Floating point arithmetic

Condition of a problem vs stability of an algorithm

Summary

References

- This talk is largely based on Chapters 1 and 2 of Nick Higham's "Accuracy and Stability of Numerical Algorithms", 2nd ed., 2002, SIAM.
- It also uses examples from presentations by Higham, Kahan and Cody.

Learning objectives

What I want you to know. . .

The basic principles of floating point arithmetic and its limitations

What I want you to be able to do. . .

Be aware of the pitfalls! Be critical of your simulation results!

Computers cannot handle π

Computers have a limited memory

- So a limited, truncated approximation of the number must be stored
- They cannot store
  3.141 592 653 589 793 238 462 643 383 279 502 884 197 169 399 375 105 8.

  only 3.141 592 653 589 793

Computers cannot handle π

Rounding and rules for arithmetic are needed

A computer number system requires rules

- to represent numbers
- to do arithmetic with these representations (in the best possible way)
- to round the results

Computers cannot handle π

Rounding and rules for arithmetic are needed

A computer number system requires rules

- to represent numbers
- to do arithmetic with these representations (in the best possible way)
- to round the results

Accidents happen. . .

The “classics”...

The Patriot Missile Software problem

- During operation Desert Storm, a Patriot missile failed to intercept: 28 casualties
- Aim was based on “counting” tenths of seconds + rounding

Hours	Seconds	Calculated (seconds)	Inaccuracy (seconds)	Approx. shift (meters)
0	0	0	0	0
1	3600	3599.9966	0.0034	7
8	28800	28799.9725	0.0275	55
20	72000	71999.9313	0.687	137
48	172800	172799.8352	0.1648	330
72	259200	259199.7528	0.2472	494
100	360000	359999.65678	0.3433	687

The “classics”...

The Vancouver Stock Exchange

- Jan 1982: index was set at 1000
- Nov 1983: index was 520!!!

but the exchange was doing well

The “classics”...

The Vancouver Stock Exchange

- Jan 1982: index was set at 1000
- Nov 1983: index was 520!!!

but the exchange was doing well

Rounding errors were the source of the error

- rounding mode: round down to 3 digits (123.45 → 123)
- lots of small errors resulting in large error

Correct recalculation: index doubled!

A floating point number system

A floating point number system is a subset of real numbers

$$F \subset \mathbb{R}$$

A number $y \in F$ is represented as

$$y = \pm m \times \beta^{e-t}$$

A floating point number system

F is characterized by integer parameters

β: The base (2, 10, 16)

t: The precision

e: The exponent, with $e_{min} \leq e \leq e_{max}$

m: The significant or mantissa, an integer with $0 \leq m \leq \beta^{t-1}$

Properties

- For each non-zero $y \in F$: $m \geq \beta^{t-1}$ (system is “normalized”)
- $y = 0$ is a special case (cannot be normalized)
- Range of non-zeros: $\forall y \in F : \beta^{e_{\min}-1} \leq |y| \leq \beta^{e_{\max}} \left(1 - \beta^{-t}\right)$

A simple example

A small, primitive FP system

The system [2]

$$\beta = 2,\ t = 3,\ e_{min} = -1,\ e_{max} = 3$$

has a range

$$\forall y \in F_0 : \beta^{e_{\min}-1} \leq |y| \leq \beta^{e_{\max}} \left(1 - \beta^{-t}\right)$$

$$\rightarrow \forall y \in F_0 : 0.25 \leq |y| \leq 7$$

[2] (see Higham 2002, p.36)

A simple example

Python code

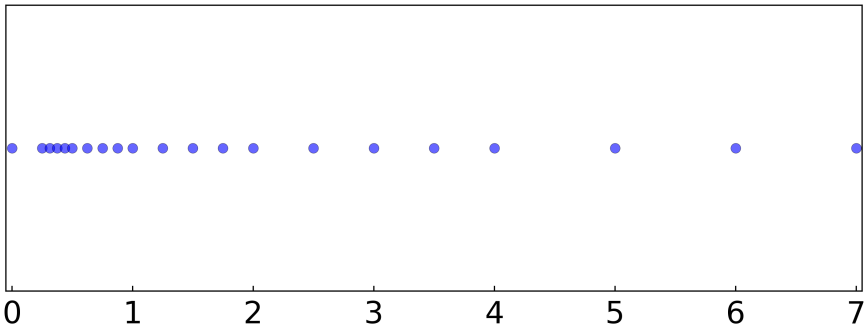
```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt

fp = [0.0]
emin = -1
emax = 3
beta = 2
t = 3
for e in range(emin,emax+1):
    for m in range(beta**(t-1),beta**t):
        fp.append(m*beta**(e-t))

print(fp)

plt.style.use('seaborn-notebook')
p1 = plt.figure(figsize=(12,4))
plt.scatter(fp,np.zeros_like(fp),s=60,alpha=0.6)
plt.tick_params(axis='x',labelsize=24)
plt.tick_params(left='off',right='off')
plt.tick_params(labelleft='off',top='off')
plt.xlim(-0.05,7.05)
plt.ylim(-0.005,0.005)
plt.savefig('FP_Ex1.png',dpi=300,bbox_inches = 'tight')
```

FP system



[0.0, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1.0, 1.25,

1.5, 1.75, 2.0, 2.5, 3.0, 3.5, 4, 5, 6, 7]

A simple example

Python code

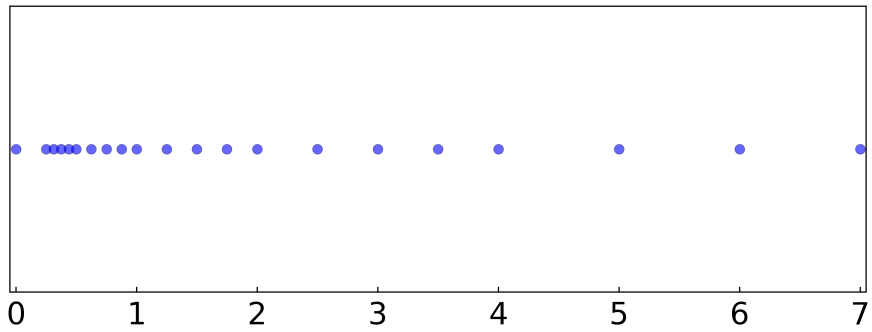
```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt

fp = [0.0]
emin = -1
emax = 3
beta = 2
t = 3
for e in range(emin,emax+1):
    for m in range(beta**(t-1),beta**t):
        fp.append(m*beta**(e-t))

print(fp)

plt.style.use('seaborn-notebook')
p1 = plt.figure(figsize=(12,4))
plt.scatter(fp,np.zeros_like(fp),s=60,alpha=0.6)
plt.tick_params(axis='x',labelsize=24)
plt.tick_params(left='off',right='off')
plt.tick_params(labelleft='off',top='off')
plt.xlim(-0.05,7.05)
plt.ylim(-0.005,0.005)
plt.savefig('FP_Ex1.png',dpi=300,bbox_inches = 'tight')
```

FP system



[0.0, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1.0, 1.25,

1.5, 1.75, 2.0, 2.5, 3.0, 3.5, 4, 5, 6, 7]

Spacing increases by factor of 2 at each power of 2

An important quantity: ϵ_M

The machine epsilon

$$\epsilon_M = \beta^{1-t}$$

- Distance between 1.0 and the next floating point number in F
- The spacing between a normalized FP number x and an adjacent normalized FP number is at least $\beta^{-1}\epsilon_M|x|$ and at most $\epsilon_M|x|$

Extension of F

Subnormal numbers

- subnormal numbers: $y = \pm m \times \beta^{e_{\min}-t},\ 0 < m < \beta^{t-1}$
- numbers with the minimal exponent and which are not normalized
- have less digits of precision than normalized numbers

Special cases

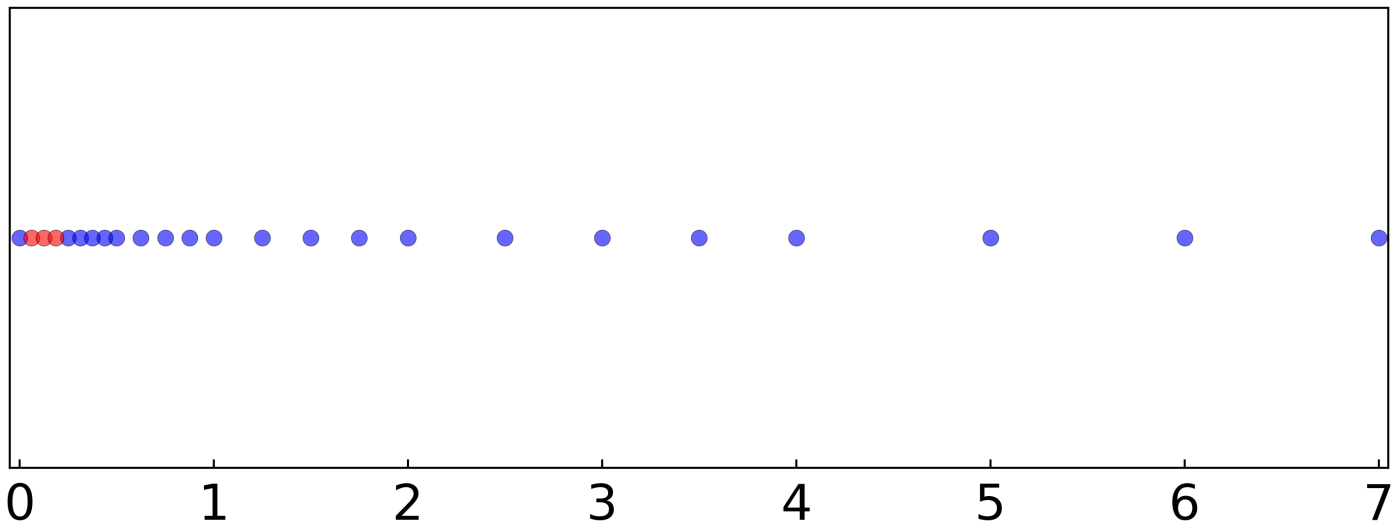
- Smallest positive normalized number: $\lambda = \beta^{e_{\min}-1}$
- Smallest positive non-normalized number: $\mu = \beta^{e_{\min}-t} = \lambda\epsilon_M$
- Subnormal numbers fill the gap between 0 and λ and are equally spaced with a spacing $\lambda\epsilon_M$

Previous example

For our system $F : \beta = 2,\ t = 3,\ e_{\min} = -1,\ e_{\max} = 3$ we have

- $\lambda = 2^{-2} = 0.25$ and $\mu = 2^{-4} = 0.0625$
- Hence, the subnormal numbers are

$$[0.0625, 0.125, 0.1875]$$



A real life example

IEEE Standard 754 (1985) is a binary floating point system describing two formats (single and double)

For double precision:

- $\beta = 2, t = 53, e_{\text{min}} = -1021, e_{\text{max}} = 1024$
- smallest positive number = $2.225\,073\,858\,507\,20 \times 10^{-308}$
- largest positive number = $1.797\,693\,134\,862\,32 \times 10^{308}$
- machine epsilon $\epsilon_M = 2.220\,446\,049\,250\,313 \times 10^{-16}$

Mapping of real numbers to FP numbers

Rounding

- Let $G \subset \mathbb{R}$ be a FP system with no limit on the exponent e.
- For $x \in \mathbb{R}$ then $fl(x)$ denotes the element of G nearest to x
- Rounding: mapping $x \rightarrow fl(x)$

However, real FP systems are limited in exponent. . .

Mapping of real numbers to FP numbers

Rounding

- For $x \in \mathbb{R}$, find the nearest element $fl(x) \in G$
- If $|fl(x)| > \max\{|y| : y \in F\}$: $fl(x)$ **overflows**
- If $0 < |fl(x)| < \min\{|y| : 0 \neq y \in F\}$: $fl(x)$ **underflows**
  - If subnormal numbers are included in F: gradual underflow (loss of digits)

Approximation error for the mapping $x \rightarrow fl(x)$

Fundamental theorem of Floating Point number systems

If $x \in \mathbb{R}$ lies in the range of F, then

$$fl(x) = x(1+\delta), \qquad |\delta| < u \equiv \frac{1}{2}\beta^{1-t}$$

The quantity u is named the unit roundoff

This implies that any real number in the range of F can be approximated by an element of F with a relative error no larger than u!

For IEEE double precision: $u = 1.110\,223\,024\,625\,156\,5 \times 10^{-16}$

Floating point arithmetic

Representation of a number

if $x \in \mathbb{R}$ lies in the range of F then

$$fl(x) = x(1+\delta),\ |\delta| < u$$

Standard model for arithmetic

$$fl(x\ op\ y) = (x\ op\ y)(1+\delta),\ |\delta| < u,\ \text{where } op = +, -, *, /$$

IEEE Standard 754 (1985)

Some “extras”

- all operations done as if in infinite precision, then rounded
- IEEE Standard 754 is a closed system thanks to the inclusion of NaN (Not a Number) and $\pm\infty$
- IEEE Standard 754 raises exceptions (can be caught by user) in case of “strange operations”

Exception type	Example	Default result
Invalid operation	$0/0, 0 \times \infty, \sqrt{-1}$	NaN
Overflow		$\pm\infty$
Divide by zero	finite nonzero$/0$	$\pm\infty$
Underflow		Subnormal numbers
Inexact	Whenever $fl(x\ op\ y) \neq x\ op\ y$	Correctly rounded results

Limitations of floating point arithmetic

Not all numbers are present in F

F is a real subset of $\mathbb{R}$

Some classic rules of arithmetic are lost

- $a * (b + c) = a * b + a * c$
- $(a * b) * c = a * (b * c)$
- $\left(\sqrt{x}\right)^2 = x$

Counting to six in floating point arithmetic

$$1 = \quad 2 - 1 \rightarrow 1.000\,000\,000\,000\,00$$

$$2 = \quad \left(\frac{1}{\cos\left(100\pi + \frac{\pi}{4}\right)} \right)^2 \rightarrow 2.000\,000\,000\,000\,111$$

$$3 = \quad 3 \cdot \frac{\tan(\arctan(10000))}{10000} \rightarrow 2.999\,999\,999\,997\,161\,8$$

$$4 = \quad \left(\left(\cdots \left(\sqrt{\sqrt{\cdots \sqrt{4}}} \right)^2 \cdots \right)^2 \right)^2 \rightarrow 1.000\,000\,000\,000\,00$$

$$5 = \quad 5 \cdot \left(\frac{(1+\exp(-100))-1}{(1+\exp(-100))-1} \right) \rightarrow \mathrm{NaN}$$

$$6 = \quad \frac{\log(\exp(6000))}{1000} \rightarrow +\infty$$

Classical test problems

$\sin(22)$

sin is calculated by argument reduction and then an approximation developed for the reduced interval

- Bring the argument in the interval $[-\pi/2, +\pi/2]$ by using the fact that $\sin(x - 2\pi) = \sin(x)$
- First problem: representation of π is also FP!
- Second problem: hard if x is close to a multiple of π

```
>>> import numpy as np
>>> exact = -8.8513092904038759216 9e-3
>>> print('%30.16e'%np.sin(22))
       -8.8513092904038762e-03
```

Floating point arithmetic

Summary

- Computers cannot represent $\mathbb{R}$
- Only a (very) limited subset $F \subset \mathbb{R}$
- Representation + rules of arithmetic = FP system
- IEEE 754 implements this rigorously

Don’t be afraid, be aware. . .

Floating point arithmetic

Summary

- Computers cannot represent $\mathbb{R}$
- Only a (very) limited subset $F \subset \mathbb{R}$
- Representation + rules of arithmetic = FP system
- IEEE 754 implements this rigorously

Don’t be afraid, be aware. . .

And if floating point is a limiting factor in your simulations, you can always resort to symbolic math packages (Maple,. . .) or special multi-precision libraries (GMP).

The conditioning of a problem

Conditioning

“The sensitivity of the solution of a problem to perturbations in the data”

The conditioning of a problem

Conditioning

“The sensitivity of the solution of a problem to perturbations in the data”

A classic example: the Wilkinson polynomial

$$P_W(x) = \prod_{i=1}^{20}(x-i) = x^{20} - 210\,x^{19} + \ldots + 20!$$

Roots: $x_i = 1, 2, \ldots, 20$

Wilkinson's polynomial

```
>>> import numpy as np
>>> W = np.poly(np.linspace(1,20,20))
>>> rts = np.roots(W)
>>> for i in range(10):
...     print('{0:20.16e}  {1:20.16e}'.format(rts[i],rts[2*i+1]))
...
2.000003842284999e+01  1.899944525724052e+01
1.899944525724052e+01  1.698821520784347e+01
1.800332671569957e+01  1.495636365244276e+01
1.698821520784347e+01  1.295597520413775e+01
1.602622950542975e+01  1.098552895731637e+01
1.495636365244276e+01  8.998586495313717e+00
1.405068644326395e+01  6.999977300409713e+00
1.295597520413775e+01  5.000000230261207e+00
1.202995480081650e+01  2.999999999192100e+00
1.098552895731637e+01  9.999999999974043e-01
```

Condition number

The condition number κ

The condition number measures the (relative) change in the output for a given relative change in the input.

$$\frac{\hat{y}-y}{y} = \kappa(x)\frac{\Delta x}{x} + \mathcal{O}\left((\Delta x)^2\right)$$

$$\text{with } \kappa(x) = \left|\frac{x\, f'(x)}{f(x)}\right|$$

Condition number of linear systems

Solving a linear system

$$Ax = b$$

is trivially solved using the well-known LU-factorisation based on Gaussian elimination with pivoting:

- Compute $PA = LU$ with L lower triangular, U upper triangular and P a permutation matrix
- Solve $Ly = Pb$
- Solve $Ux = y$

For the square matrix A, a condition number $\kappa(A)$ is defined by

$$\kappa(A) = \|A\|\|A^{-1}\|$$

Condition number of linear systems

From G. Golub, F. Van Loan, “Matrix Computations”, 3rd ed., 1996, John Hopkins Press

Heuristics

1. Gaussian elimination produces a solution $\hat{x}$ with a small residual ($r = A\hat{x} - b$)
2. If the unit roundoff $u \approx 10^{-d}$ and the condition of the matrix $\kappa(A) \approx 10^{q}$, then Gaussian elimination produces a solution $\hat{x}$ that has about $d - q$ correct digits

Example

When you solve a linear system $Ax = b$ with $\kappa_A \approx 10^8$ in IEEE double precision ($d = 16$), you will get a solution with about 8 digits correct.

Classic “bad” example polynomial interpolation

Find the interpolating polynomial $p(x) = \sum_{j=0}^{n} a_j x^j$ for a set of data points (x_i, f_i)

This can be written as a linear problem of the form

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}$$

The matrix is named a Vandermonde matrix

Classic “bad” example polynomial interpolation

Find the interpolating polynomial $p(x) = \sum_{j=0}^{n} a_j x^j$ for a set of data points (x_i, f_i)

$$Va = f$$

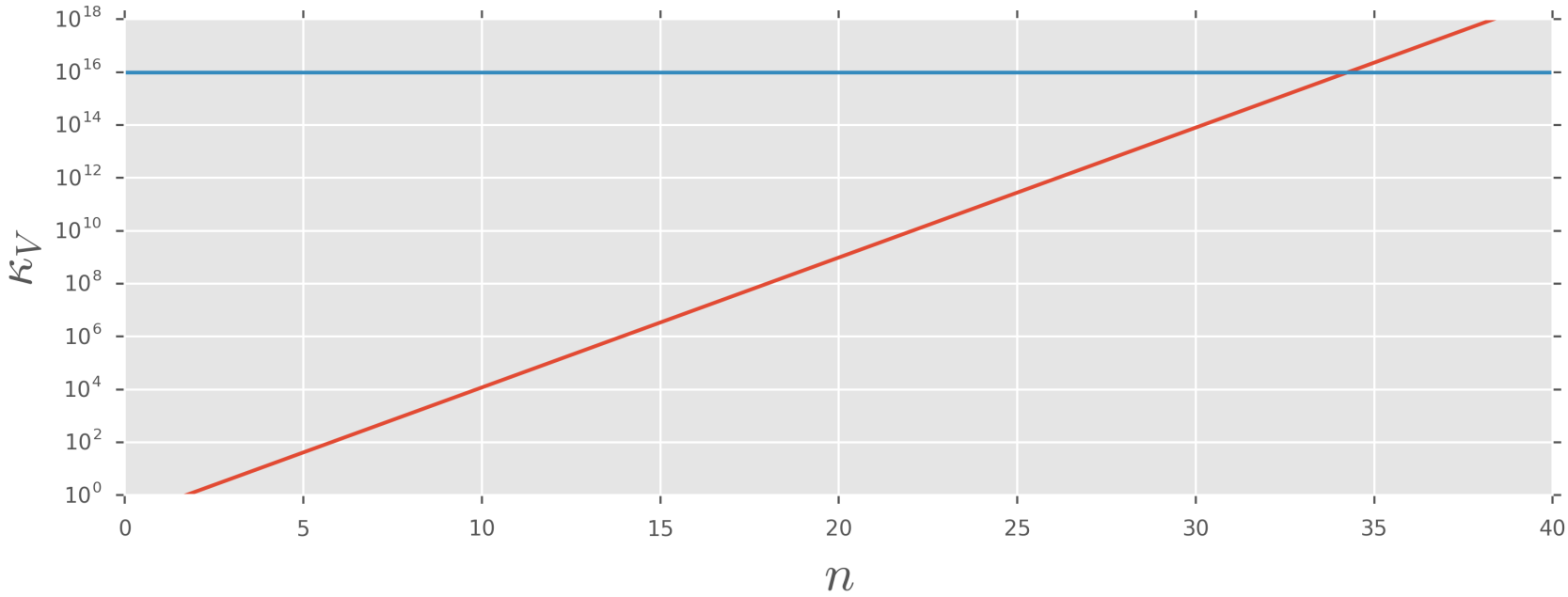
The matrix V has a notoriously bad condition number which depends on the interpolation points.

For example, for equidistant interpolation points in $[-1, +1]$, one can show that

$$\kappa_V \sim \pi^{-1} e^{-\pi/4} (3.1)^n$$

Classic “bad” example polynomial interpolation

Find the interpolating polynomial $p(x) = \sum_{j=0}^{n} a_j x^j$ for a set of data points (x_i, f_i)



Stability of an algorithm

Stability

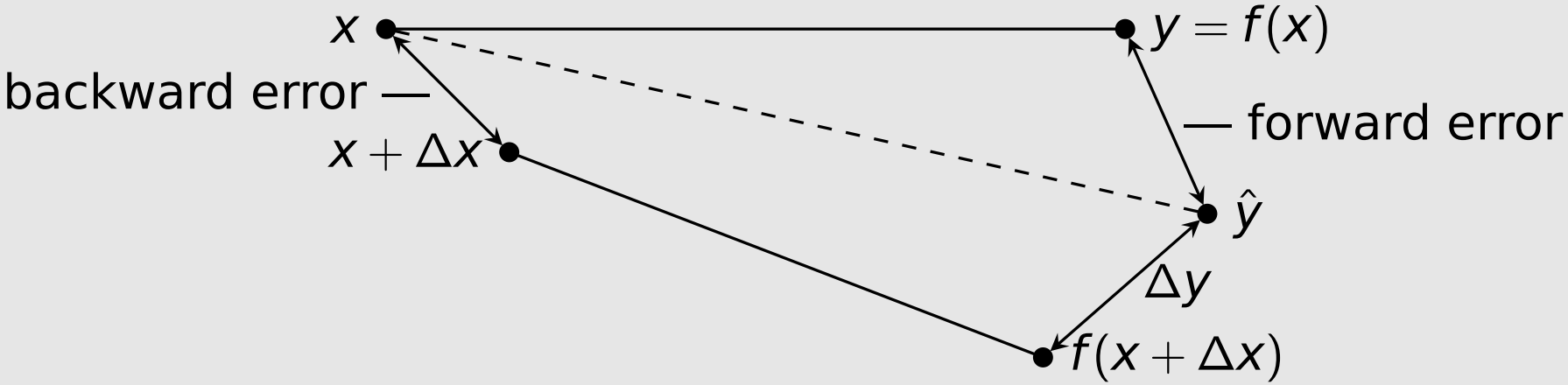
Numerical stability measures the difference between the exact solution to a problem and the calculated solution where the calculation suffers from rounding errors

Note: in other fields of scientific computing the term stability is used in other contexts (for example in solving PDEs)

A stable algorithm must

- be stable in every step
- avoid error growth

Stability related to mixed forward-backward error



A simple example: the quadratic equation

Solving the quadratic equation

$$ax^2 + bx + c = 0$$

if $a \neq 0$, there are two roots given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Stable or unstable?

A simple example: the quadratic equation

Solving the quadratic equation

$$ax^2 + bx + c = 0$$

if $a \neq 0$, there are two roots given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Unstable!

Suppose $b^2 \gg |4ac| \rightarrow \sqrt{b^2 - 4ac} \approx |b|$

A simple example: the quadratic equation

Unstable approach

$$ax^2 + bx + c = 0$$

if $a \neq 0$, there are two roots given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Stable approach

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

$$x_2 = \frac{c}{a \cdot x_1}$$

A second example

$$(e^x - 1)/x$$

A second example

$$(e^x - 1)/x$$

Approach 1

```
def fun1(x):

    if (x == 0.0):
        f = 1.0
    else:
        f = (np.exp(x)-1.0)/x
    return f
```

A second example

$$(e^x - 1)/x$$

Approach 1

```
def fun1(x):

    if (x == 0.0):
        f = 1.0
    else:
        f = (np.exp(x)-1.0)/x
    return f

xlist = np.logspace(-1,-15,15)

for x in xlist:
    print('{0:5.2e} {1:20.16e} {2:20.16e}'.format(x,fun1(x),fun2(x)))
```

Approach 2

```
def fun2(x):
    y = np.exp(x)
    if (y == 1.0):
        f = 1.0
    else:
        f = (y - 1)/np.log(y)
    return f
```

A second example

$$(e^x - 1)/x$$

Python output

```
1.00e-01 1.0517091807564771e+00 1.0517091807564762e+00
1.00e-02 1.0050167084167949e+00 1.0050167084168058e+00
1.00e-03 1.0005001667083846e+00 1.0005001667083415e+00
1.00e-04 1.0000500016671410e+00 1.0000500016667082e+00
1.00e-05 1.0000050000069649e+00 1.0000050000166667e+00
1.00e-06 1.0000004999621837e+00 1.0000005000001666e+00
1.00e-07 1.0000000494336803e+00 1.0000000500000017e+00
1.00e-08 9.9999999392252903e-01 1.0000000050000000e+00
1.00e-09 1.0000000827403710e+00 1.0000000005000000e+00
1.00e-10 1.0000000827403710e+00 1.0000000000500000e+00
1.00e-11 1.0000000827403710e+00 1.0000000000050000e+00
1.00e-12 1.0000889005823410e+00 1.0000000000005000e+00
1.00e-13 9.9920072216264089e-01 1.0000000000000500e+00
1.00e-14 9.9920072216264089e-01 1.0000000000000051e+00
1.00e-15 1.1102230246251565e+00 1.0000000000000004e+00
```

More examples are numerous

Recurrence relations

- Many “special” functions in mathematical physics (Legendre functions, Bessel functions, . . .) can be calculated using three term recurrences

$$\mathcal{F}_{k+1}(x) = a(x,k)\mathcal{F}_k + b(x,k-1)\mathcal{F}_{k-1}, \quad k = 1, 2, \ldots$$

- Recurrences have maximal and minimal solutions (think about Bessel’s $I_k(x)$ and $K_k(x)$ functions)
- You cannot calculate the minimal solution using a forward recurrence ($k = 1, 2, \ldots$)
- You need a trick (backward recurrence and re-normalisation)

Introduction

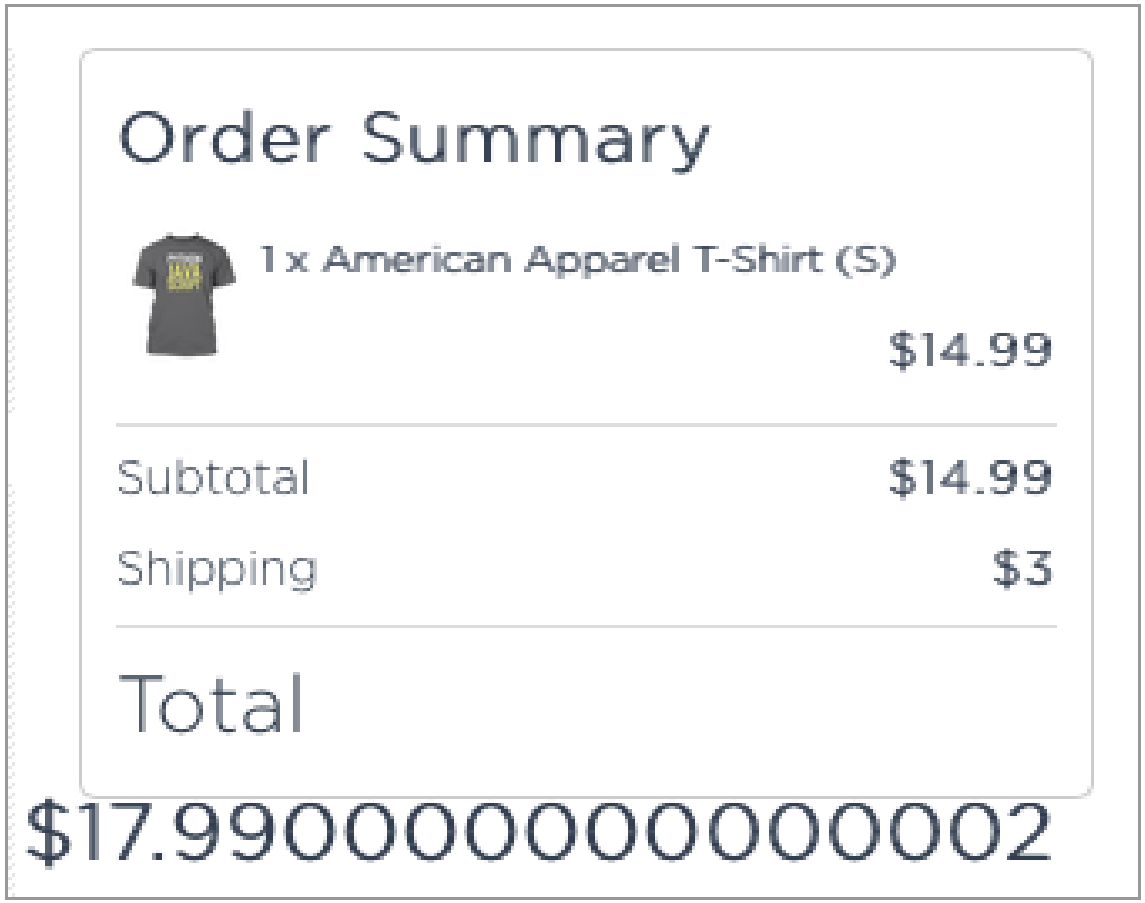
Representing floating point numbers

Floating point arithmetic

Condition of a problem vs stability of an algorithm

Summary

Summary



Be aware !!!