# Chapter 1: Numbers and Expressions

:

"All the mathematical sciences are founded on relations between physical laws and laws of numbers, so that the aim of exact science is to reduce the problems of nature to the determination of quantities by operations with numbers"

James Clerk Maxwell 1856

# Topics:

- numbers like 2, 3.75, $\pi$ and $\sqrt{19}$
- evaluation of expressions like

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

- representation and approximation of numbers and expressions
- computational errors, can they be avoided or at least controlled?

# 1.1 Numbers

# Numbers for Computations

- integers, rational, real and complex numbers

$$\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

- evaluate expressions
- for solution of equations
  - linear systems of equations $\mathbb{Q}, \mathbb{R}, \mathbb{C}$
  - polynomial equations $\mathbb{C}$
- for continuous functions: $\mathbb{R}$ and $\mathbb{C}$

# Definition: Ring

- A ring $R$ is a set with two binary operations $+$ and $*$ with the following properties
- $(R, +)$ is an abelian group which satisfies, for all $a, b, c \in R$:
  - $a + (b + c) = (a + b) + c$, *associative law*
  - $a + b = b + a$, *commutative law*
  - there exists $0 \in R$ such that $a + 0 = a$
  - there exists $-a$ such that $a + (-a) = 0$
- $(R, *)$ is a monoid where for all $a, b, c \in R$:
  - $a * (b * c) = (a * b) * c$
  - there exists $1 \in \mathbb{R}$ such that $1 * a = a$
- distributive law

$$(a + b) * c = a * c + b * c, \quad a * (b + c) = a * b + a * c$$

# Rings in Computation

- all the number sets considered are rings including $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ and $\mathbb{C}$
- the sets of functions considered are rings including
  - continuous functions
  - polynomials
- set of square $n$ by $n$ matrices with elements from a ring is a ring
- the arithmetic laws lead to efficient expression evaluation
- the distributive law is the basis for fast algorithms like fast matrix multiplication, the FFT but also for machine learning and dynamic programming
  - note that the number of multiplications is 2 in $a * b + a * c$ but 1 in $a * (b + c)$

# $\mathbb{Q}$ and $\mathbb{R}$

- $\mathbb{Q}$ is a countable subset of $\mathbb{R}$
- $\mathbb{R}$ is used for theory but (subsets of) $\mathbb{Q}$ used for actual computations
- $\mathbb{Q}$ is dense in $\mathbb{R}$, i.e. $\forall x \in \mathbb{R}, \epsilon > 0, \exists u \in \mathbb{Q}$ :

$$|x - u| \leq \epsilon$$

# Decimal and binary fractions

- for computations we consider subsets of $\mathbb{Q}$ for a fixed positive $B \in \mathbb{Z}$

$$\mathbb{Q}_B = \{p\,/B^k \mid p, k \in \mathbb{Z}, k \geq 0\}$$

- here we only consider $B = 10$ or $B = 2$:
  - decimal fractions $\mathbb{Q}_{10}$ used for manual computations
  - binary fractions $\mathbb{Q}_2$ implemented in computer hardware
- we use the decimal point, e.g.,

$$44.78 = \frac{4478}{10^2}$$

# Fractions

$$\mathbb{Z} = \mathbb{Q}_1 \subset \mathbb{Q}_2 \subset \mathbb{Q}_{10} \subset \mathbb{Q}$$

- $\mathbb{Q}_2$ (and $\mathbb{Q}_{10}$) is a dense subset of $\mathbb{Q}$ and thus of $\mathbb{R}$. In particular, each real number can be written as an infinite decimal or binary fraction.
- The sets of fractions $\mathbb{Q}_B$ are all rings and contain the integers

**Proposition** $\mathbb{Q}_2 \subset \mathbb{Q}_{10}$

*Proof.*

- $x \in \mathbb{Q}_2$, thus
$$x = \frac{p}{2^k}$$
for some integers $p$ and $k \geq 0$.
- consequently
$$x = \frac{5^k p}{10^k} \in \mathbb{Q}_{10}$$

# 1.2 Representation of Numbers

# Simple representation of Integers

- small integers
  - counts: $|, ||, |||, ||||, \ldots$, each number is the cardinality of a set
  - digits: $0, 1, 2, \ldots, 9$, each number has a symbol
  - roman numbers: $I, II, III, IV, \ldots, XXXII, \ldots$
- numbers need to be represented in order to do arithmetics
- all computers (including us) are finite

```
# implementing your own numbers in Python using strings --

x = '||||'
y = '||'

z = x + y    # concatenation is addition

print("OUTPUT:")

print('sum x + y = {}'.format(z))

n = len(z)  # conversion to ordinary integers
print('sum in decimal number system x+y = {}'.format(n))

u = 13*'|'  # conversion to our system
print('conversion of 13:  {}'.format(u))
```

```
OUTPUT:
sum x + y = ||||||
sum in decimal number system x+y = 6
```

# Representation of Integers

- computer and human number representations are similar, and of the form

$$n = \pm \sum_{k=0}^{t} n_k \, B^k$$

$B$: *basis* (humans: $B = 10$, computers: $B = 2$)

# The Polynomials which represent integers

- polynomial of degree $n$

$$p(x) = c_0 + c_1 x + \cdots + c_n x^n$$

- set $P$ of all polynomials is an infinite dimensional linear (vector) space with basis $1, x, x^2, \ldots$
- is also a ring with multiplication defined by

$$p * q\,(x) = p(x)q(x)$$

  - a ring which is also a vector space is called an *algebra*

- for representing the positive integer $n = n_0 + n_1 B + \cdots n_t B^t$ choose the polynomial

$$p(x) = n_0 + n_1 x + \cdots + n_t x^t$$

  and one has

$$n = p(B)$$

- example $n = 739$ and $B = 10$:

# Representation of Rationals

- rationals $x \in \mathbb{Q}$ are representated as pairs of integers $(p, q)$ and written as
$$x = \frac{p}{q}$$
- uniqueness is achieved by choosing the $\gcd(p, q) = 1$ (use Euclid's algorithm)
- as rational numbers are ratios of integers, and each integer is represented by a polynomial and a basis $B$, each rational number $x \in \mathbb{Q}$ is represented by a a rational function
$$r(x) = \frac{p(x)}{q(x)}$$
and
$$x = r(B)$$

```
# Rational numbers in Python

from fractions import Fraction

x = Fraction(16, -10)

print("OUTPUT:")

print(x)

y = Fraction('3/7')
print(y)

z = Fraction('1.2341')
print(z)
```

```
OUTPUT:
-8/5
3/7
12341/10000
```

# Representation of decimal and binary (and other) fractions

## standard integer format

- any $q \in Q_B$ is of the form

$$q = \frac{n}{B^k}$$

  for some integers $n$ and $k$
- example – approximation of $1/3$:

$$\frac{333}{1000}$$

- uses the rational function

$$r(x) = \frac{n_0 + n_1 x + \cdots n_t x^t}{x^k}$$

## scientific format

- any $q \in Q_B$ is of the form

$$q = (n_t + n_{t-1}B^{-1} + \cdots + n_0 B^{-t})B^e$$

- example – approximation of $1/3$:

$$0.333$$

- uses the rational function

$$f(x0) = (n_t + n_{-1}x^{-1} + \cdots + n_0 x^{-t})\, x^e$$

where $e = t - k$

# Representation of Real Numbers

- real numbers $x \in \mathbb{R}$ are represented as (potentially infinite) power series in the basis

$$x = \pm \sum_{j=-\infty}^{t} n_j B^j$$

  again, the basis $B = 10$ is used in human computation and $B = 2$ is used by computers

- the digits $n_j \in \{0, \ldots, B-1\}$
- real numbers need to be approximated

# Representation of Complex Numbers

- complex numbers $z \in \mathbb{C}$ are represented as pairs of reals

$$z = x + iy$$

- addition like vectors
- complex multiplication
- conjugate complex
- imaginary unit $i$

```
## Complex numbers in Python

z = 4.0 + 5.0j

print("OUTPUT:")

print("Re(z) = {zr:g}, Im(z) = {zi:g}"
      .format(zr=z.real,zi=z.imag))

I = 1j     # sqrt(-1)

print("I**2 = {}".format(I**2))
```

```
OUTPUT:
Re(z) = 4, Im(z) = 5
I**2 = (-1+0j)
```

# Floating Point Numbers – approximations of real numbers

The set of floating point numbers with $t$ digits to base $B$ is

$$\mathbb{F}_B(t) = \{x = \pm \sum_{j=1}^{t} c_j B^{-j+e} \mid e \in \mathbb{Z}, c_j \in \mathbb{Z}_B, c_1 \neq 0\} \cup \{0\}$$

where $\mathbb{Z}_B = \{0, \ldots, B-1\}$.

$$\mathbb{F}_B(t) \subset \mathbb{Q}_B \subset \mathbb{Q} \subset \mathbb{R}$$

- $\mathbb{F}_B(t)$ is *not*
  - a ring
  - dense in $\mathbb{Q}$
- motivation: subset of $\mathbb{F}_B(t)$ with $e = e_{\text{min}}, \ldots, e_{\text{max}}$ is computationally feasible
- challenge: floating point arithmetic has to be (re-)defined

```
## Floating point numbers in Python

x = 0.7
print("OUTPUT:")
print("computer approximation \nx= {0:2.53g}".format(x))

OUTPUT:
computer approximation
x= 0.69999999999999955591079014993738383054733276367187 5
```

# The IEEE standard 754 floating point numbers

- $B = 2$ and $t = 53$, each number is stored in 64 bit
- special numbers are: 0, $\pm\infty$, some non-normalised numbers, NaNs
- the exponents $e = -1022, \ldots, 1023$
- the standard also specifies details about the arithmetic

# Representation of floating point numbers

- representation of any floating point number

$$x = \pm 0.d_1 d_2 \dots d_t \cdot B^e$$

  $B$: *base*, $d_j \in \{0, \dots, B-1\}$: *digits*, $e$: *exponent*, *number of digits* $t$

- written as a sum

$$x = sB^e \sum_{j=1}^{t} d_j B^{-j}$$

  $s = +1, -1$ sign

- normalised $d_1 \neq 0$
- $\mathbb{F}_<(B, t)$ denotes floating point numbers with $t$ digits in base $B$ (here we allow any $e \in \mathbb{Z}$, in practice it is a finite range, see notes)

# IEEE 754 standard on representation

- most commonly used system today: IEEE double precision

| number system | base B | number of digits t | exponent range |
|---|---|---|---|
| IEEE double precision | 2 | 53 | [-1022, 1023] |
| IEEE single precision | 2 | 24 | [-126, 127] |

- Note: there are many other formats, both decimal and binary

# Effect of finite exponent

There is a smallest (srictly positive) floating point number

- normalised: $x_{\mathsf{min}} = B^{e_{\mathsf{min}}-1}$
- denormalised: $B^{e_{\mathsf{min}}-t}$

There is a largest (positive) floating point number *
$x_{\mathsf{max}} = B^{e_{\mathsf{max}}} - B^{e_{\mathsf{max}}-t}$

Errors occur when computations exceed these limits

- *underflow* occurs for $0 < x < x_{\mathsf{min}}/2$
- *overflow* occurs when numbers exceed $x_{\mathsf{max}}$ We ignore these types of errors in the remainder by allowing $e \in \mathbb{Z}$

# Questions

# 1.3 Real numbers in Python

# Decimal module

The module decimal implements the arithmetic we usually do by hand. All the standard arithmetic can be used and, if necessary, the results will be rounded to a number of digits which can be changed by setting the context parameter prec. However, the available functions are limited and the operations may take longer than with the built in data types.
Note that the numbers have to be input as strings (as otherwise Python would automatically round to floating point numbers).

```
import decimal as dec

x = dec.Decimal('0.6')
y = dec.Decimal('0.5999999999')  # 10 significant digits
z = x - y
```

```
print("x = {0}, y = {1}, x-y = {2}\n".format(x,y,z))
print("representation of x: {!r}\n".format(x))
print("type of x: {}\n".format(type(x)))

print("timing")
%timeit(x-y)

x = 0.6, y = 0.5999999999, x-y = 1E-10

representation of x: Decimal('0.6')

type of x: <class 'decimal.Decimal'>

timing
101 ns ± 0.729 ns per loop (mean ± std. dev. of 7 runs, 10(
```

# Python's default real numbers

Python uses 64 bit floating point numbers with 53 binary significant digits per default for real numbers. The arithmetic with these numbers is implemented in hardware and is very fast. However, even simple numbers like 0.6 cannot be exactly represented as floating point numbers of this type and thus one gets rounding errors. The effect is even worse when one substracts two numbers which are close and looses a significant amount of digits. This is called *cancellation*.

The usage of floating point numbers is illustrated below:

```
## Default floating point numbers

x = 0.6
y = 0.5999999999  # 10 significant decimal digits
z = x - y

print("x = {0}, y = {1}, x-y = {2}\n".format(x,y,z))
print("representation of x: {!r}\n".format(x))
print("type of x: {}\n".format(type(x)))
```

```
x = 0.6, y = 0.5999999999, x-y = 1.000000082740371e-10

representation of x: 0.6

type of x: <class 'float'>
```

```
print("printing the result with some more digits:\n")
print("   x = {0:3.100g}\n   y = {1:3.100g}\n   x-y = {2:3.
```

```
xstring = "{:3.100g}".format(x)
xmystring = "{:3.100g}".format(x-y)
print("significant decimal digits:   x: {0},     x-y: {1}\n
```

```
print("timing")
%timeit(x-y)
```

```
printing the result with some more digits:

   x = 0.5999999999999999777955395074968691915273666381835
   y = 0.5999999989999999695215024075878318399190902709960
   x-y = 1.000000082740370999090373516082763671875e-10

significant decimal digits:   x: 53,     x-y: 39

timing
40.7 ns ± 0.389 ns per loop (mean ± std. dev. of 7 runs, 1
```

# Real numbers in numpy

The numerical package numpy has three major floating point number systems: float16, float32 and float64. They are useful for the development of resource critical applications using lower accuracy arithmetic whic is supported by some hardware including some graphics boards. In particular using these types is in principle a first way to save energy as the costliest operations are data transfers and using the types below on can thus cut costs by a factor of up to four.

The significant (binary) digits of the three numpy types are

- float16 has 11
- float32 has 24
- float64 has 53

Note that only 10, 23 and 52 of these bits are stored as it is assumed that the first significant bit is always 1. The accuracy and timing is illustrated below. On my laptop the timings were the same, actually, the 64 bit version was the fastest! The reason for this might be that the actual arithmetic is still done in the hardware floating point unit (using 64 or even higher accuracy) and the result is then rounded. As a direct conversion from float16 and float32 to decimal is not supported we first convert these two types to float64 (which can be done without error). Also note that the difference is less than the error for float16 and one gets then 100 percent error.
**NB As we are computing with lower accuracy, we have changed the problem a bit compared to above!**

```
## numpy floating point numbers -- print out using the dec

import decimal as dec
import numpy as np

# three numpy types

x16 = np.float16(0.6)
y16 = np.float16(0.59999)
z16 = x16 - y16

x32 = np.float32(0.6)
y32 = np.float32(0.59999)
z32 = x32 - y32

x64 = np.float64(0.6)
y64 = np.float64(0.59999)
z64 = x64 - y64
```

```
# exact values
xex = dec.Decimal("0.6")
yex = dec.Decimal("0.59999")
zex = xex - yex

print("   x16 = {0}\n   x32 = {1}\n   x64 = {2}\n".format(
```

```
   x16 = 0.60009765625
   x32 = 0.60000002384185791015625
   x64 = 0.59999999999999977795539507496869191527366638183
```

```
print("relative rounding errors:\n")
e16 = (dec.Decimal(float(x16)) - xex)/xex
e32 = (dec.Decimal(float(x32)) - xex)/xex
e64 = (dec.Decimal(x64) - xex)/xex
print("  for x:     float16 :   {:3.2g},   float32 :   {:3

em16 = (dec.Decimal(float(z16)) -zex)/zex
em32 = (dec.Decimal(float(z32)) -zex)/zex
em64 = (dec.Decimal(float(z64)) -zex)/zex
print("  for x-y:   float16 :   {:3.2g},       float32 :
```

```
relative rounding errors:

  for x:     float16 :   0.00016,   float32 :   4.0e-8,   f
  for x-y:   float16 :    -1,       float32 :   0.0014,   f
```

```
print("timing  - not much difference at this level")
print("* float16")
%timeit(x16-y16)
print("* float32")
%timeit(x32-y32)
print("* float64")
%timeit(x64-y64)
```

```
timing  - not much difference at this level
* float16
87.7 ns ± 0.776 ns per loop (mean ± std. dev. of 7 runs, 1(
* float32
75.5 ns ± 0.608 ns per loop (mean ± std. dev. of 7 runs, 1(
* float64
76.7 ns ± 0.836 ns per loop (mean ± std. dev. of 7 runs, 1(
```

# npmath multiple precision

This module does provide floating point with choosable (binary) accuracy. We will choose 113 binary digits which is the standard for quadruple (128 bit) arithmetic.
In order to compute the exact numbers we first convert the npmath number to a string (note that we need to use 113 bit decimal precision to get the exact result). Then we convert this string to a Decimal. Note that multiple precision operations are substantially slower than the floating point ones.
Looking at the printout it seems that only the first 30 or so decimal digits are accurate and the later ones are wrong. Thus without loosing much accuracy, one could set these later digits to zero. However, we should remember, that the numerical approximation is binary number with 113 digits and the number is accurate to all the binary digits. If one now changes any of the later decimal digits and then rounds again to the nearest binary number on typically gets a larger error.

```
import mpmath as mpm

prec = 113
mpm.mp.prec = prec # set precision to quadruple

## Default floating point numbers -- print out using the d

x = mpm.mpf('0.6')
y = mpm.mpf('0.5999999999')  # 10 significant decimal digi
z = x - y

xex = dec.Decimal("0.6")                  # exact values ..
yex = dec.Decimal("0.5999999999")
zex = xex - yex

xdec = dec.Decimal(mpm.nstr(x,prec))  # frist convert to s
ydec = dec.Decimal(mpm.nstr(y,prec))
zdec = dec.Decimal(mpm.nstr(z,prec))
```

```
print(" x = {0}\n y = {1}\n x-y = {2}\n".format(xdec, ydec,

print("errors:\n")
e = (xdec - xex)/xex
print("relative rounding error of x :   {:3.2g}".format(e))
em = (zdec -zex)/zex
print("relative rounding error of x-y:   {:3.2g}\n".format(

 x = 0.599999999999999999999999999999807407005561276414
 y = 0.599999999000000000000000000000063405484118044044
 x-y = 9.999999999999999999997440015214432369822629110

errors:

relative rounding error of x :   -3.2e-35
relative rounding error of x-y:   -2.6e-25
```

```
print("timing")
%timeit(x-y)
```

```
timing
1.66 µs ± 9.08 ns per loop (mean ± std. dev. of 7 runs, 10(
```

# Other options

One can use other C data types and can also get access to the 80 or 128 bit accuracy of the hardware processor. Especially for running on GPUs one may also use the data types these processors use natively. For our purposes the above methods will be sufficient.

# 1.4 relative error

In computations we often cannot determine the exact value of some real value $x \in \mathbb{R}$. Instead, some value $\tilde{x} \in \mathbb{R}$ is computed which hopefully is close to $x$ in some sense. This closeness or accuracy of $\tilde{x}$ we describe with statements like $\tilde{x}$ *is accurate to three (decimal) digits*. This concept of accuracy is modelled by the concept of *relative error*.

**Definition:** An approximation $\tilde{x}$ of a real number $x$ has a relative error $\delta$ if

$$\tilde{x} = (1 + \delta)x.$$

The value of $\delta$ is thus

$$\delta = \frac{\tilde{x} - x}{x}$$

in the case where $x \neq 0$. Note that $\delta$ is well defined for any number $\tilde{x}$.

For example, consider the approximation of 3.14 for $\pi$.

```
from math import pi
delta = (pi - 3.14)/pi
print("relative error of 3.14: {}".format(delta))
```

```
relative error of 3.14: 0.0005069573828972128
```

In practice, we will not know the (exact) value $x$. Thus the value of $x$ is uncertain. In error analysis we aim to determine bounds for relative error $\delta$ of $\tilde{x}$ based on the properties of the computations performed.

For example, we know that the floating point arithmetic used in Python has a 53 bit mantissa (t=53) and a base $B = 2$. From this one can see that the relative error occurring in optimal rounding satisfies

$$|\delta| \leq \epsilon = \frac{1}{2B^{t-1}}$$

which in our case is

```
B = 2.0
t = 53
epsilon = 0.5/B**(t-1)
print("bound of relative error {}".format(epsilon))
```

```
bound of relative error 1.1102230246251565e-16
```

We now check how well a number we input is rounded in Python. We consider $x = 3.45$ and use decimal arithmetic. With this we get the exact value of the difference $\tilde{x} - x$.

```
from decimal import Decimal
x = Decimal("3.45")
xtilde = 3.45
delta = (Decimal(xtilde) - x)/x
print("relative error delta = {}".format(delta))
```

```
relative error delta = 5.14886040405869699906611788 1E-17
```

As this is less than the bound, we may also conclude that the rounding is optimal in this case.

Given a relative error, one can now determine the number of (significant) digits of $\tilde{x}$ are accurate by using the 10 based logarithm. For our example of $\pi$ we get

```
from math import pi, log
delta = (pi - 3.14)/pi
digits = round(-log(abs(delta))/log(10))
print("accurate digits of 3.14: {}".format(digits))
```

```
accurate digits of 3.14: 3
```

For the rounding error bound we get the number of accurate digits we expect in that case to be

```
B = 2.0
t = 53
epsilon = 0.5/B**(t-1)
digits = round(-log(abs(epsilon))/log(10))
print("accurate digits after rounding: {}".format(digits))
```

```
accurate digits after rounding: 16
```

# 1.5 Decimal Rounding

# The decimal floating point numbers with two digits $\mathbb{F}_{10}(2)$

- Any positive (normalised) $x \in \mathbb{F}_{10}(2)$ is of the form

$$x = \frac{n}{100} 10^e, \quad n = 10, \ldots, 99, \ e \in \mathbb{Z}$$

- There are 91 such floating point numbers between 0.1 and 1:

$$S = \{0.10, 0.11, 0.12, \ldots, 0.99, 1.0\}$$

- Every number $x \in S$ which is less than one has a *successor*

$$\operatorname{succ} x = x + 0.01$$

- We will also use the set of 90 *midpoints* between the floating point numbers

$$M = \{0.105, 0.115, 0.125, \ldots, 0.985, 0.995\}$$

and $M \not\subset \mathbb{F}_{10}(2)$

# Rounding

A *rounding function* $\phi : \mathbb{R} \to \mathbb{F}_{10}(2)$ has the following properties

- $\phi(x) = x$ for $x \in \mathbb{F}_{10}(2)$
- if $x \leq y$ then $\phi(x) \leq \phi(y)$ (monotonicity)
- $\phi(-x) = -\phi(x)$
- $\phi(10x) = 10\,\phi(x)$

It follows from the first two properties that any value $\phi(x)$ is either equal to the next lower or next higher floating point number, for example

$$\phi(0.12456) \in \{0.12, 0.13\}$$

The third and fourth property lets us extend the definition of $\phi$ from the interval $[0.1, 1]$ to the whole set of real numbers $\mathbb{R}$. For example, one has

$$\phi(124.56) = 100\,\phi(0.12456)$$

applying the fourth property twice

Examples of rounding functions include

- *truncation* where

$$\phi(0.x_1x_2x_3\ldots) = 0.x_1x_2$$

  and thus $\phi(0.1256) = 0.12$

- *rounding towards zero*
  - $\phi(0.x_1x_2x_3\ldots) = 0.x_1x_2$ if $x_3 \in \{0, 1, 2, 3, 4\}$
  - $\phi(0.x_1x_2x_3) = 0.x_1x_2$ if $x_3 = 5$ (midpoints)
  - $\phi(0.x_1x_2x_3x_4\ldots) = 0.x_1x_2 + 0.01$ if $x_3 = 5$ and $x_i > 0$ for some $i > 4$

and thus $\phi(0.1256) = 0.13$, $\phi(0.125) = 0.12$ and $\phi(0.124) = 0.12$

- *rounding used in most computers* is the same as rounding towards zero except that the second condition for the midpoints is replaced by two cases:
  - $\phi(0.x_1x_2x_3) = 0.x_1x_2$ if $x_3 = 5$ and $x_2$ is even
  - $\phi(0.x_1x_2x_3) = 0.x_1x_2 + 0.01$ if $x_3 = 5$ and $x_2$ is odd

thus $\phi(0.125) = 0.12$ but $\phi(0.135) = 0.14$
this condition corrects for the bias towards zero

Finally, a rounding function $\phi$ is *optimal* if it minimises the *rounding error* $|\phi(x) - x|$, i.e., if

$$|\phi(x) - x| \leq |y - x|, \quad \text{for all } y \in \mathbb{F}_{10}(2)$$

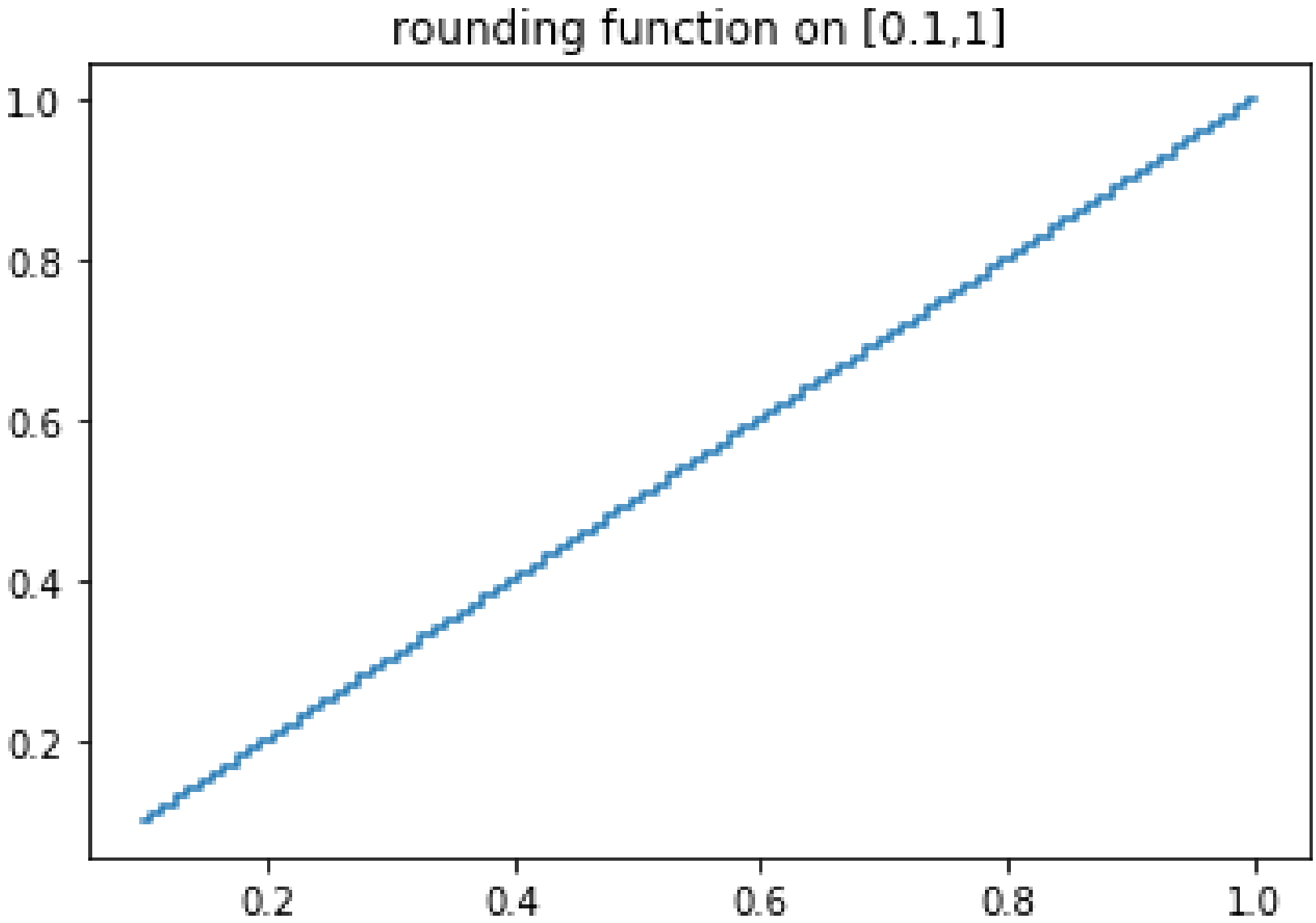
Truncation is not optimal, but both rounding towards zero and the rounding used in most computers are optimal.
Note that the rounding function used in computers rounds to a different set $\mathbb{F}_2(53)$, however, it uses the same tie-breaking strategy for the midpoints.

In the following we plot the graph of the rounding function $\phi$ both for $[0.1, 1]$ and for $[0.1, 10]$. Note that on each intervall $[10^{e-1}, 1^e]$ the rounding function is a step function with constant steps at the midpoints between the floating point numbers. The height of the step is proportional to $10^e$.

```
# optimal rounding functions

%matplotlib inline
from decimal import Decimal, getcontext
getcontext().prec = 3
from pylab import plot, title, loglog
t = 2
h = Decimal('0.1')**t
x = Decimal('0.1')
xg = [x,]
yg = [x,]
```

```
for i in range(9*10**(t-1)):
    xg.append(x+h/2) # midpoint
    yg.append(x)
    xg.append(x+h/2) # midpoint
    yg.append(x+h)
    xg.append(x+h)
    yg.append(x+h)
    x += h
```

```
title('rounding function on [0.1,1]')
plot(xg,yg);
```



```
xg += [10*x for x in xg]
yg += [10*y for y in yg]
```

```
title('rounding function on [0.1,10]')
loglog(xg,yg);
```

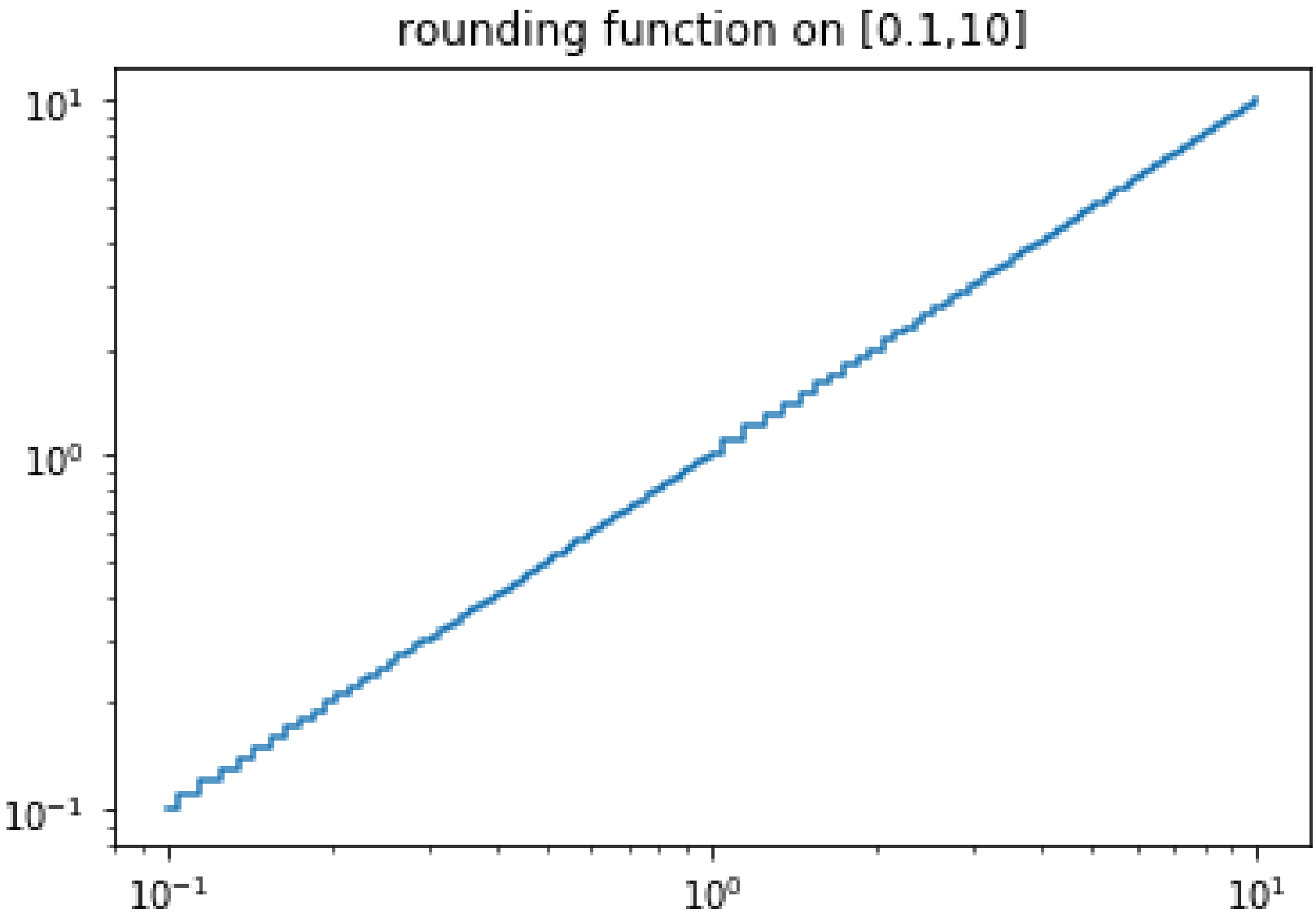



Figure 2: png

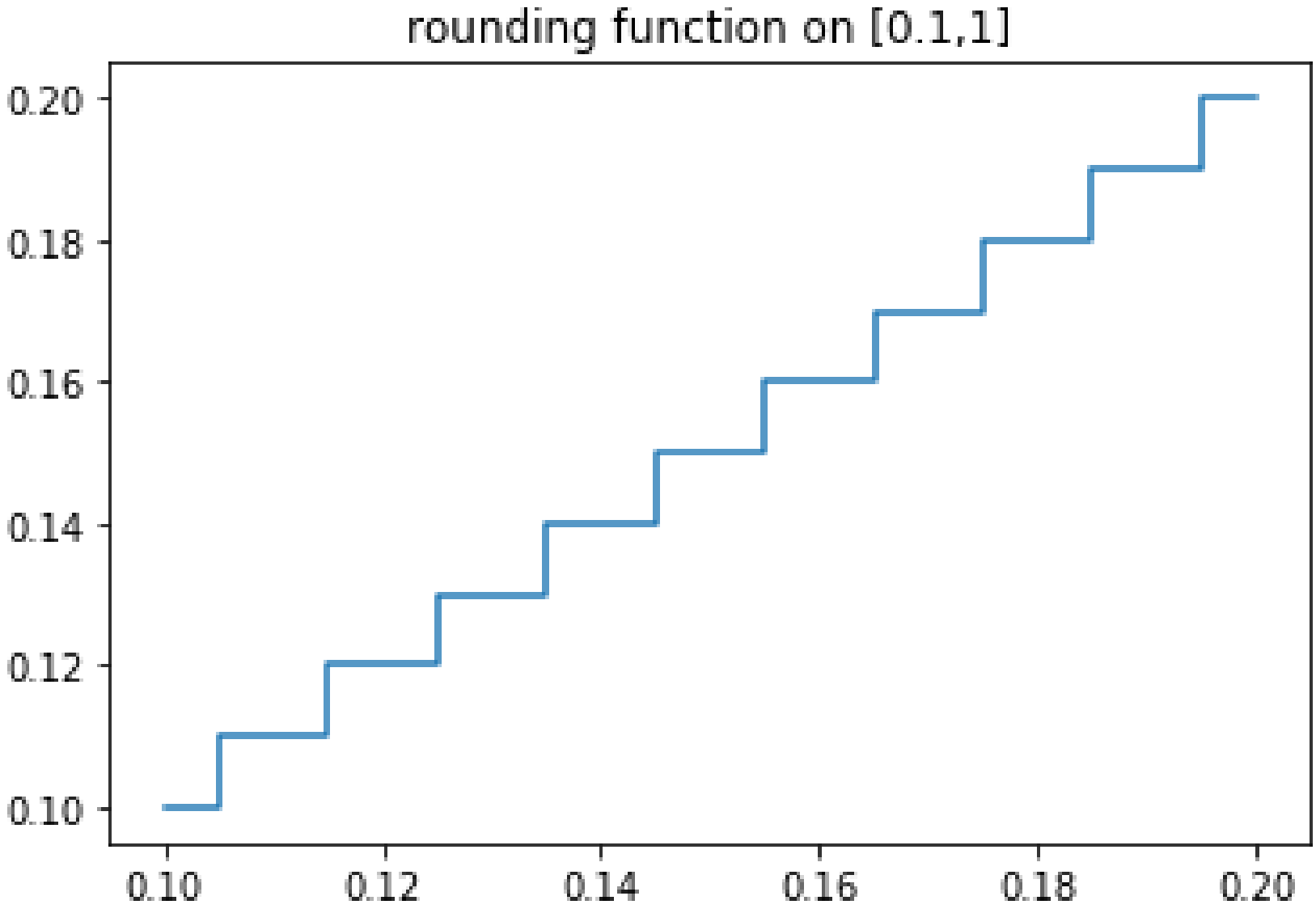
# Rounding errors

In the following plots we have a closer look at the rounding function and the absolute and relative rounding errors.

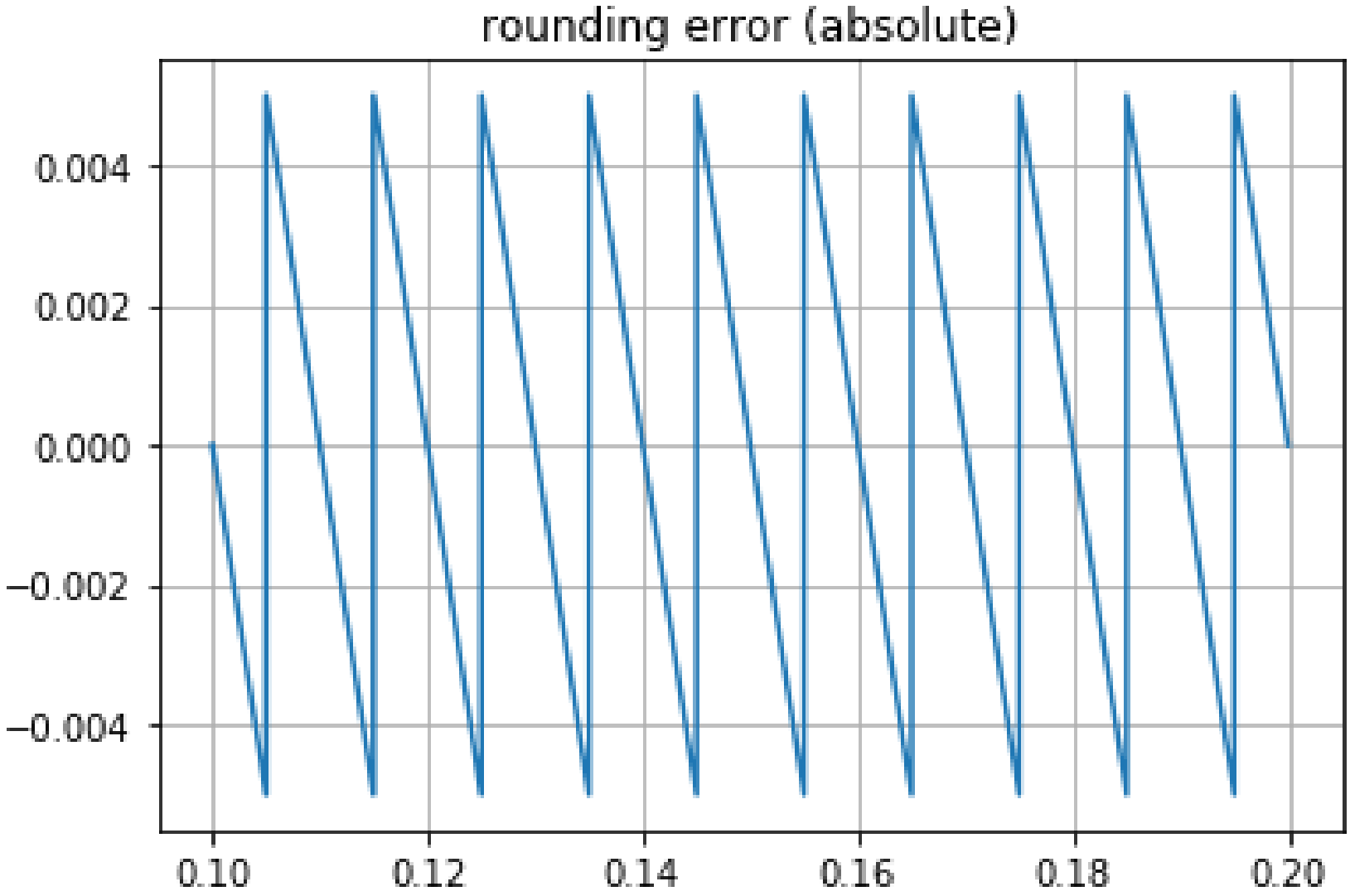
```
# rounding  function and error

%matplotlib inline
from decimal import Decimal, getcontext
getcontext().prec = 6
from pylab import plot,  title, loglog, grid
h = Decimal('0.01')
hg = h/20
x = Decimal('0.1')
y = Decimal('0.1')
xg = [x,]
yg = [y,]
nx = 10
```

```
for k in range(nx):
    for i in range(10):
        x += hg
        xg.append(x)
        yg.append(y)
    y += h
    xg.append(x)    # double up midpoint
    yg.append(y)
    for i in range(10):
        x += hg
        xg.append(x)
        yg.append(y)
```

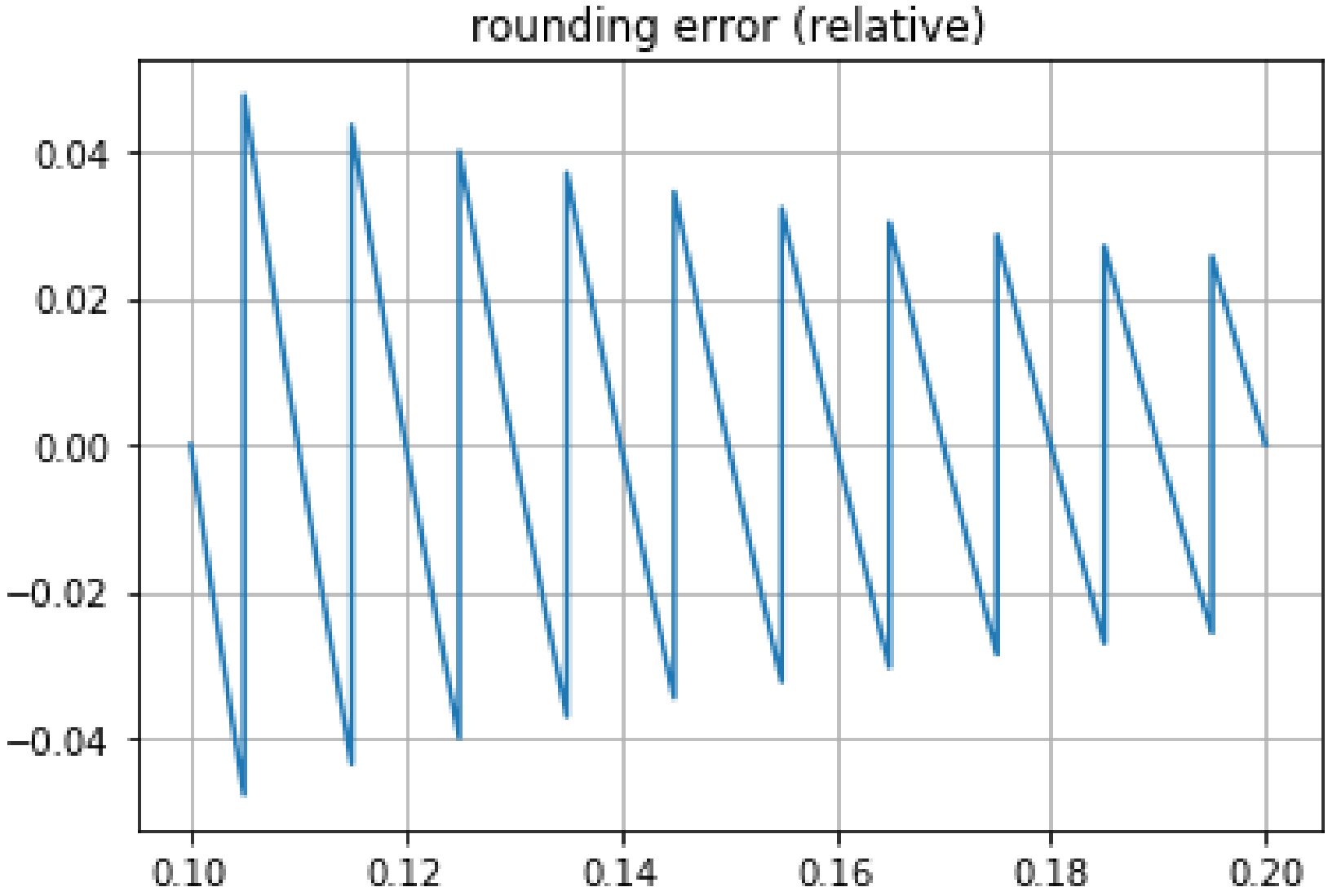
```
title('rounding function on [0.1,1]')
plot(xg,yg);
```



```
title('rounding error (absolute)'); grid('on');
eg = [yg[i]-xg[i] for i in range(len(xg))]
plot(xg, eg);
```



```
title('rounding error (relative)'); grid('on');
erg = [(yg[i]-xg[i])/xg[i] for i in range(len(xg))]
plot(xg,erg);
```



One can see that the maximal relative rounding error occurs at the first midpoint $x = 0.105$. This is rounded to $\tilde{x} = 0.1$ and the error is thus $\tilde{x} - x = -0.005$. The absolute value of the maximal relative error is then

$$\delta = \left| \frac{\tilde{x} - x}{x} \right| = \frac{0.005}{0.105} = 0.0476.$$

This is close to the upper bound of $0.5B^{-t+1}$ (with $B = 10$ and $t = 2$) given in the lectures and the bound gets better for larger mantissa sizes $t$.

Suggestion: Study the plots and derive a formal proof that the maximum of the rounding error is indeed obtained in the first mid point.

# 1.6 Rounding

# floating point numbers

- recall that we are computing in the number system $\mathbb{F}_B(t)$ with $B = 2$ or $B = 10$
- a typical $x \in \mathbb{F}_B(t) \setminus \{0\}$ is of the form

$$x = \pm \left( \frac{x_1}{B} + \frac{x_2}{B^2} + \cdots + \frac{x_t}{B^t} \right) B^e$$

  with exponents $e \in \mathbb{Z}$ and digits $x_k \in \{0, \ldots, B-1\}$ and $x_1 \neq 0$
- in short we write this as

$$x = 0.x_1 x_2 \ldots x_t \, B^e$$

  - for example one has $x = -0.4521 \cdot 10^5$

# Rounding function $\phi(x)$

- mapping
$$\phi : \mathbb{R} \to \mathbb{F}_B(t)$$
  - there are many different rounding functions including truncation
    - example: $\phi(0.346) = 0.34$
  - set of floating point numbers discrete, thus rounding function piecewise constant
- the rounding error for some $x \in \mathbb{R}$ is then
$$\epsilon(x) = \phi(x) - x$$
  - error piecewise linear, not continuous
  - rounding errors of complicated expressions can be hard to predict
  - $\phi(x)$ monotone: $\phi(x) \leq \phi(y)$ if $x \leq y$

# Questions

- When do you use rounding?
- In projects, when estimating costs and time?
- How many digits do you usually require?
- What happens if we do not round?

# Optimal Rounding

- A rounding function $\phi$ is called *optimal* if

$$|\phi(x) - x| \leq |y - x|, \quad y \in \mathbb{F}_B(t)$$

- Truncation is not optimal: e.g., for $\mathbb{F}_{10}(2)$ one has for $\phi =$ truncation

$$\phi(0.4563) = 0.45$$

but

$$|0.45 - 0.4563| = 0.0063 > |0.46 - 0.4563| = 0.0037$$

**Proposition** *An optimal rounding satisfies*

$$\phi(x) - x = \delta x$$

*where* $|\delta| \leq 0.5B^{-t+1}$

*Proof.*

- if $x \in \mathbb{F}_B(t)$ then $\phi(x) = x$ by optimality
- if $x \notin \mathbb{F}_B(t)$ then $\phi(x) \in \{x_1, x_2\}$ where $x_i \in \mathbb{F}_B(t)$ are the two closest numbers to $x$
- by optimality $|\phi(x) - x| \leq |x_2 - x_1|/2$
- If $x > 0$ one has $x = \sum_{j=1}^{\infty} c_j B^{-j+e}$ and

$$x_1 = \sum_{j=1}^{t} c_j B^{-j+e} < x_2 = x_1 + B^{-t+e}$$

  and so $(x_2 - x_1)/2 = 0.5B^{-t+e}$ and similar for $x < 0$
- Normalisation: $B^{-1+e} \leq c_1 B^{-1+e} \leq x$ and thus

$$\frac{x_2 - x_1}{2x} \leq \frac{0.5B^{-t+e}}{B^{-1+e}} = 0.5B^{-t+1}$$

- Optimal rounding is not unique, for example, if $x = 0.745$ both both 0.74 and 0.75 are optimal (in $\mathbb{F}_{10}(2)$)
  - a common choice is in this case to select the number with even least significant digit, i.e., 0.74

# examples

1. floating point decimal numbers in $\mathbb{F}_{10}(2)$:
   - $\phi(3.452) = 3.5$
   - $\phi(0.675) = 0.68$
   - $\phi(1/9) = 0.11$
2. floating point binary number in $\mathbb{F}_2(3)$:
   - $\phi(3.1875) = 3$ as $3.1875 = 0.110011_2 \cdot 2^2$ and $\phi(0.110011_2 \cdot 2^2) = 0.11_2 \cdot 2^2 = 3$

# A property of floating point rounding

**Lemma**
If $\phi_0 : \mathbb{R} \to \mathbb{Z}$ and $\phi : \mathbb{R} \to \mathbb{F}_B(t)$ are rounding functions (with consistent rounding of midpoints) and $e$ is the exponent of $x \in \mathbb{R}$ (normalised) then

$$\phi(x) = \phi_0(B^{t-e}x)/B^{t-e}$$

*for proof use:* $\quad \{B^{t-1}, \ldots, B^t - 1\} = \{B^{t-e}y \mid y \in \mathbb{F}_B(t)\} \in \mathbb{Z}$

```
# Rounding to Fl_2(t) in Python:

def roundfl2(number, ndigits=1):
    import math
    (xm, xe) = math.frexp(number)
    xr = round(xm*2.0**ndigits)/2.0**ndigits
    return math.ldexp(xr, xe)

roundfl2(3.1875, ndigits=4)

3.25
```

# using Python decimal module for rounding in $\mathbb{F}_{10}(t)$

- Python *decimal* which implements floating point numbers
- we use this module to implement a decimal rounding function
- output in floating point thus additional error

```
# decimal rounding of binary floating point numbers

def roundfl10(x, t=1):
    from decimal import Context
    return float(Context(prec=t).create_decimal(x))

roundfl10(3.1875, t=3)
```

```
3.19
```

# Applications of Rounding to elementary unary and binary functions

As $\mathbb{F}_B(t)$ is not a ring, we need to approximate all arithmetic operations and an optimal approximation of these operations is

- for example, we replace the sum $x + y$ by $\phi(x + y)$, and the same for multiplications
- any unary function evaluations are also done using rounding, e.g. replace $\sin(x)$ by $\phi(\sin(x))$

In order to assess the error caused through rounding one uses the proposition above to get

- for the binary function evaluations: $(1 + \delta_1)(x + y)$
- for unary function evaluations: $(1 + \delta_2)\sin(x)$

(of course, the $\delta_i$ are not the same but in an ideal case, they are bounded by the same constant)

- the $\delta_i$ characterise the relative rounding error which occurs when the functions are done on a computer

# arithmetic operations in $\mathbb{F}_B(t)$

- difference in $\mathbb{F}_B(t)$ exact only in exceptional circumstances, a notable case is where $x$ and $y$ are very close as in this case the difference of $x$ and $y$ is also in $\mathbb{F}_B(t)$
- the product $x * y$ of two numbers in $\mathbb{F}_B(t)$ is in $\mathbb{F}_B(2t)$
- the quotient will typically be a floating point number with an infinite number of digits
- IEEE 754 standard suggests that best approximation using rounding should be used to implement arithmetic operations

*thus replace any* $x \circ y$ *by* $\phi(x \circ y)$

# properties of approximate arithmetic

- commutative law holds for addition and multiplication in $\mathbb{F}_B(t)$

$$\phi(xy) = \phi(yx)$$

- associative law for addition does not hold
  - e.g. $\phi(x + \phi(y + z)) \neq \phi(\phi(x + y) + z)$
  - e.g. in $\mathbb{F}_{10}(3)$

$$\phi(\phi(1.32{+}0.254){+}0.392) = 1.96 \neq 1.97 = \phi(1.32{+}\phi(0.254{+}0.392))$$

- neither associative law for multiplication nor the distributive law hold

# simple functions

- IEEE 754 also requires that simple functions like exp, sin etc are implemented such that they are the rounded version of the exact function
- For example, we may define $\sin_{\mathbb{F}}(x) := \phi(\sin(x))$ where $\phi : \mathbb{R} \to \mathbb{F}$ is a rounding function

# Questions

The next section deals with numeric expressions.

- What is the significance of such expressions, where are they used?
- What is the math behind the expressions?
- Can you think of anything related to your studies, work and life where such expressions play a role?

# 1.7 error analysis of expressions

- recall: floating point numbers have only a finite fixed number of digits in mantissa
  - consequence: computers need to round almost every arithmetic operation and function evaluation
- most real numbers and even rational numbers (like $1/5$) are not floating point numbers
  - consequence: computers have to round almost all inputs
- the resulting *rounding errors* are unavoidable and occur in every computation

In the following we analyse these errors
Consider, for example the evaluation of

$$f(x) = 2\sin(x_1 x_2) + x_3$$

where

$$x_1 = 3.57, \quad x_2 = 0.0723, \quad \textit{and} \quad x_3 = 1.0.$$

Evaluating this on your computer gives

```
from math import sin
2*sin(3.57*0.0723) + 1.0
```

```
1.5105091672725943
```

# Step 1: rewrite expression as sequence of simple assignments

In a first step we rewrite the expression to be evaluated as a sequence of simple expressions of the form

$$
\begin{aligned}
u_0 &= f_0 \\
u_1 &= f_1(u_0) \\
u_2 &= f_2(u_0, u_1) \\
&\dots \\
u_n &= f_n(u_0, \dots, u_{n-1}).
\end{aligned}
$$

The functions $f_k$ are either non-floating point constants, in our example 3.57 and 0.0723 but not 1.0 as this is a floating point number, or simple expressions which are evaluated with a rounding error, for example $2u_3 + 1$ but not $2u_3$ (which is evaluated exactly). In the case of our example we get

$$
\begin{aligned}
u_0 &= 3.57 \\
u_1 &= 0.0723 \\
u_2 &= u_0 u_1 \\
u_3 &= \sin(u_2) \\
u_4 &= 2u_3 + 1
\end{aligned}
$$

# Step 2: include errors

We now rewrite the algorithm including rounding errors

- formula for rounding function

$$\phi(x) = (1 + \delta)x$$

note: $\delta$ depends on $x$ but satisfies

$$|\delta(x)| \leq \epsilon$$

With the substitution one then gets

$$\begin{aligned}
u_0 &= (1 + \delta_0)\, f_0 \\
u_1 &= (1 + \delta_1)\, f_1(u_0) \\
u_2 &= (1 + \delta_2)\, f_2(u_0, u_1) \\
&\dots \\
u_n &= (1 + \delta_n)\, f_n(u_0, \dots, u_{n-1}).
\end{aligned}$$

For our **example** we get

$$
\begin{aligned}
u_0 &= (1+\delta_0)\,3.57 \\
u_1 &= (1+\delta_1)\,0.0723 \\
u_2 &= (1+\delta_2)\,u_0 u_1 \\
u_3 &= (1+\delta_3)\,\sin(u_2) \\
u_4 &= (1+\delta_4)\,(2u_3+1)
\end{aligned}
$$

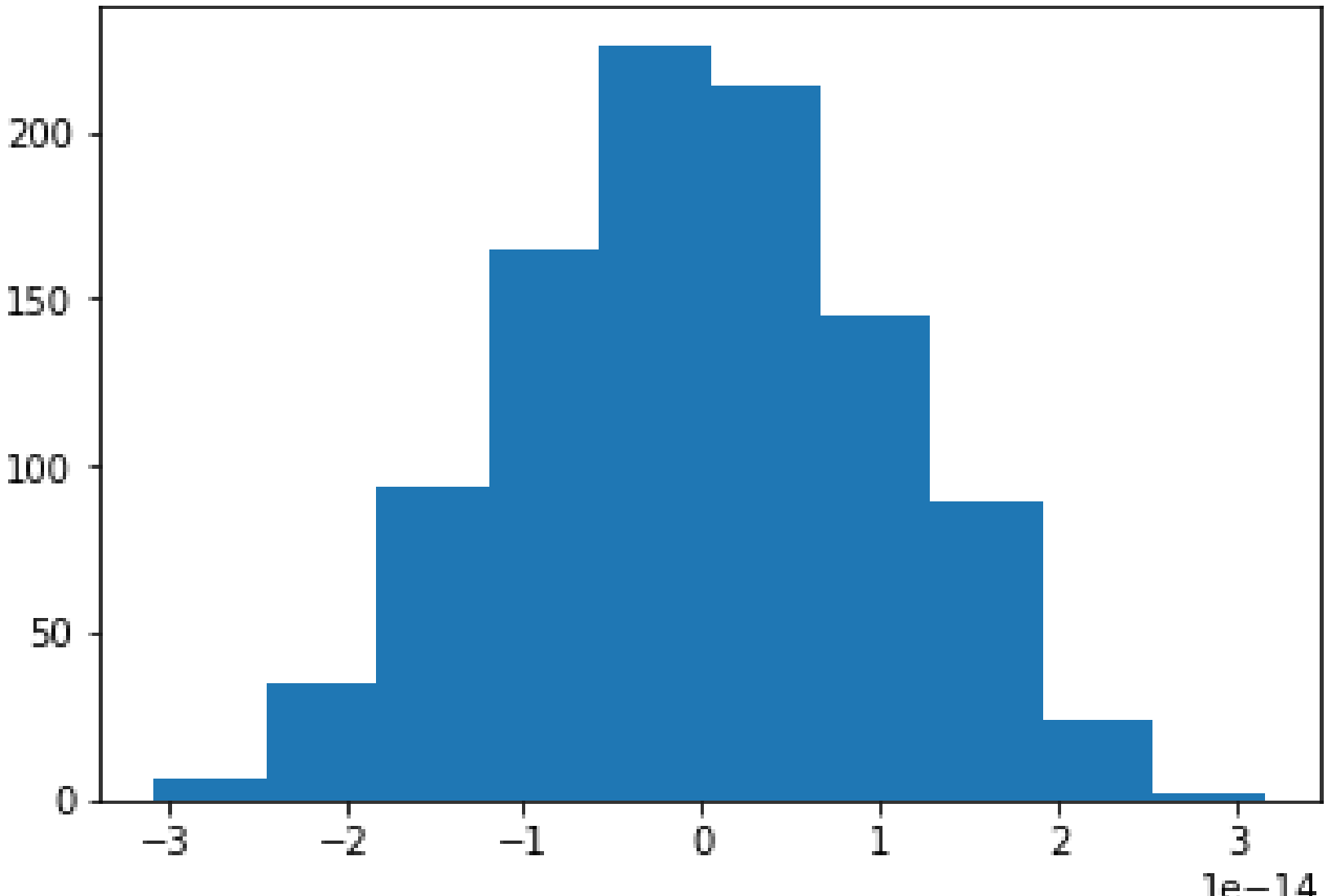
- result $u_4$ is polynomial in the $\delta_k$
- study using simulation and derive error bounds
- we have avoided dealing with the discontinuous rounding functions!

```
# study the effect of rounding errors on the result

def g(delta):

    u_0 = (1+delta[0])*3.57
    u_1 = (1+delta[1])*0.0723
    u_2 = (1+delta[2])*u_0*u_1
    u_3 = (1+delta[3])*sin(u_2)
    u_4 = (1+delta[4])*(2*u_3 + 1)

    return u_4
```

```
# compute "exact" result
import numpy as np
u_4ex = g(np.zeros(5))
print("exact result: ", u_4ex)

# simulate using random rounding errors with uniform
#   distribution for uncertain epsilon[k]
n = 1000
error = np.zeros(n)
epsi = 1e-14
for k in range(n):
    delta = epsi*(np.random.random(5)*2-1)
    error[k] = g(delta) - u_4ex
```

```
exact result:  1.51050916727
```

```
%matplotlib inline
import pylab as plt
plt.hist(error);
```



# Revision ideas:

- use above approach to analyse simple expressions like $a * b$ or $a + b + c$
- take a code you might have and include rounding errors to study their effect on the result
- any suggestions on how to automatically include rounding errors?

# 1.8 bounding the error of expressions

# modelling expressions with simple bivariate functions

- let a set of integers $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$ satisfy
  - either $i_k = j_k = 0$
  - or $j_k < i_k < k$
- let $f_1, \ldots, f_n$ be bivariate real functions defined on compact domains
  - the functions $f_k$ are either arithmetic binary operations or univariate functions
- let $u_0 = 0$ and $u_k$ be defined by the system of equations

$$u_k = f_k(u_{i_k}, u_{j_k}), \quad k = 1, \ldots, n$$

# evaluation of the expression

- these equations are thus solved (i.e. all $u_k$ computed) by substitution

$$
\begin{aligned}
u_1 &= f_1(u_0, u_0) = f_1(0, 0) \\
u_2 &= f_2(u_{i_2}, u_0) = f_2(u_{i_2}, 0), \quad i_2 \in \{0, 1\} \\
u_3 &= f_3(u_{i_3}, u_{j_3}), \quad i_3 \in \{0, 1, 2\},\ j_3 \in \{0, \ldots, i_3\} \\
&\ldots \\
u_n &= f_n(u_{i_n}, u_{j_n}), \quad i_n \in \{0, \ldots, n-1\},\ j_3 \in \{0, \ldots, i_n\}
\end{aligned}
$$

- with this we have modeled the evaluation of numerical expressions where $u_n$ is the value of the expression and the other $u_k$ intermediate results

# example $\left(-p + \sqrt{p^2 - 4\,q}\right)/2$

$$
\begin{aligned}
u_1 &= p \\
u_2 &= q \\
u_3 &= u_1^2 \\
u_4 &= u_3 - 4\,u_2 \\
u_5 &= \sqrt{u_4} \\
u_6 &= \left(-u_1 + u_5\right)/2
\end{aligned}
$$

# the same with rounding errors at every step

- now let $v_k$ be the numerical versions of $u_k$ defined by

$$v_k = (1 + \delta_k)\, f_k(v_{i_k}, v_{j_k}), \quad k = 1, \ldots, n$$

  and $v_0 = 0$
- as usual $|\delta_k| \leq \epsilon$
- the relative error of $v_k$, i.e., $(v_k - u_k)/u_k$ is denoted by $\theta_k$ so that

$$v_k = (1 + \theta_k) u_k$$

# example with rounding errors

$$
\begin{aligned}
v_1 &= (1+\delta_1)p \\
v_2 &= (1+\delta_2)q \\
v_3 &= (1+\delta_3)v_1^2 \\
v_4 &= (1+\delta_4)(v_3 - 4\,v_2) \\
v_5 &= (1+\delta_5)\sqrt{v_4} \\
v_6 &= (1+\delta_6)\,(-v_1 + v_5)\,/2
\end{aligned}
$$

# total error at every step – for multiplication and division

- recall: $f_k(x_i, x_j)$ is either an arithmetic binary operation (like sum) of $x_i$ and $x_f$ *or* a unary operation $f_k(x_i)$
- the simplest cases are multiplication and division
- for multiplication $f_k(v_i, v_j) = (1 + \theta_i)(1 + \theta_j)u_i u_j$ and so

$$v_k = (1 + \delta_k)(1 + \theta_i)(1 + \theta_j)\, u_k$$

  - multiplication:

$$\theta_k = (1 + \delta_k)(1 + \theta_i)(1 + \theta_j) - 1 \approx \theta_i + \theta_j + \delta_k$$

  - division:

$$\theta_k = (1 + \delta_k)(1 + \theta_i)/(1 + \theta_j) - 1 \approx \theta_i - \theta_j + \delta_k$$

# total error at every step – for addition and subtraction

- for addition $f_k(v_i, v_j) = (1+\theta_i)u_i + (1+\theta_j)u_j$ and so

$$v_k = (1+\delta_k)\left((1+\theta_i)\frac{u_i}{u_i+u_j} + (1+\theta_j)\frac{u_j}{u_i+u_j}\right)(u_i+u_j)$$

  - addition:

$$\theta_k = (1+\delta_k)(1+\zeta_k\theta_i + (1-\zeta_k)\theta_j) - 1 \approx \zeta_k\theta_i + (1-\zeta_k)\theta_j + \delta_k$$

    where $\zeta_k = u_i/(u_i+u_j)$

- convex combination if $u_i$ and $u_j$ have equal sign
- if different sign, error can be very large despite the fact that some times $\delta_k = 0$ in this case
- similar for subtraction

# total error at every step – for univariate function

- $f_k(v_i) = f_k((1+\theta_i)u_i)$ and so

$$\begin{aligned} v_k &= (1+\delta_k)\, f_k((1+\theta_i)u_i \\ &= (1+\delta_k)\left(1 + \frac{f_k((1+\theta_i)u_i) - f_k(u_i)}{f_k(u_i)}\right) u_k \\ &= (1+\delta_k)\,(1+\zeta_k\theta_i)u_k \end{aligned}$$

where $\zeta_k = \frac{f_k((1+\theta_i)u_i) - f_k(u_i)}{\theta_i f_k(u_i)}$ and

$$|\zeta_k| \leq \frac{L_k|u_i|}{|f_k(u_i)|}$$

if $L_k$ is Lipschitz constant of $f_k$

- relative error of $v_k$ is then

$$\theta_k = (1+\delta_k)(1+\zeta_k\theta_i) - 1 \approx \zeta_k\theta_i + \delta_k$$

# relative errors for example

$$
\begin{aligned}
\theta_1 &= \delta_1 \\
\theta_2 &= \delta_2 \\
\theta_3 &= (1+\delta_3)(1+\theta_1)^2 - 1 \\
\theta_4 &= (1+\delta_4)(1+\zeta_4\theta_3 - (1-\zeta_4)\theta_2) - 1 \\
\theta_5 &= (1+\delta_5)(1+\zeta_5\theta_4) - 1 \\
\theta_6 &= (1+\delta_6)(1-\zeta_6\theta_1 + (1-\zeta_6)\theta_5) - 1
\end{aligned}
$$

- homework: what are the $\zeta_k$, get bounds and obtain a bound for $\theta_6$

# stability and growth factor

- we say that the $f_k$ are **stable** for if there exists some $L > 0$ such that for all $k$ one has

$$|f_k(x_1, x_2) - f_k(y_1, y_2)| \leq L \max_i |x_i - y_i|$$

- we assume that for $k > 0$ one has $u_k \neq 0$
- then one can define a *growth factor*

$$\rho = \max\{|u_j|/|u_k| \mid j < k\}$$

# a simple global error bound

**Proposition** Let $\alpha = (1+\epsilon)L\rho$ where $L$ be as defined above, $\rho$ be the growth factor then

$$v_k = (1+\theta_k)u_k$$

where

$$|\theta_k| \leq \left(\frac{\alpha^{k+1}-1}{\alpha-1}\right)\epsilon$$

**proof.**

- induction
- first one has

$$v_1 = (1 + \delta_1)u_1$$

and thus $\theta_1 = \delta_1$ and $|\theta_1| = |\delta_1| \leq \epsilon$

- then

$$\begin{aligned} v_{k+1} &= (1 + \delta_{k+1}) f_{k+1}(v_{i_{k+1}}, v_{j_{k+1}}) \\ &= (1 + \theta_{k+1}) u_{k+1} \end{aligned}$$

where

$$\theta_{k+1} = \delta_{k+1} + (1 + \delta_{k+1}) \frac{f_{k+1}(v_{i_{k+1}}, v_{j_{k+1}}) - f_{k+1}(u_{i_{k+1}}, u_{j_{k+1}})}{u_{k+1}}$$

- the (absolute value of the) first term is bounded by $\epsilon$ and for the second term one has for some $0 < i \leq k$:

$$(1+\delta_{k+1})\left|\frac{f_{k+1}(v_{i_{k+1}}, v_{j_{k+1}}) - f_{k+1}(u_{i_{k+1}}, u_{j_{k+1}})}{u_{k+1}}\right| \leq (1+\epsilon)L\frac{|v_i - u_i|}{|u_{k+1}|}$$

$$= \frac{(1+\epsilon)L|\theta_i| \cdot |u_i|}{|u_{k+1}|}$$

$$\leq L(1+\epsilon)\frac{\alpha^{i+1}-1}{\alpha-1}\rho\epsilon$$

$$\leq \frac{\alpha^{k+2}-\alpha}{\alpha-1}\epsilon$$

from which one gets

$$|\theta_{k+1}| \leq \frac{\alpha^{k+2}-1}{\alpha-1}\epsilon$$

■

# example $\left(-p+\sqrt{p^2-4\,q}\right)/2$

$$
\begin{aligned}
u_1 &= p \\
u_2 &= q \\
u_3 &= u_1^2 \\
u_4 &= u_3 - 4\,u_2 \\
u_5 &= \sqrt{u_4} \\
u_6 &= \left(-u_1 + u_5\right)/2
\end{aligned}
$$

$$\Longleftrightarrow$$

$$\mathbf{U} = \mathbf{F}(\mathbf{U}), \quad \mathbf{U} = \left(u_1, u_2, u_3, u_4, u_5, u_6\right)^T$$

# Linearized model

$$\mathbf{V} = \mathbf{F}(\mathbf{V}) + \underbrace{\beta}_{\text{abs. round error}}, \quad \mathbf{V} = (v_1, v_2, v_3, v_4, v_5, v_6)^T$$

- Assume that the $f_k$ are **continuously differentiable** so that

$$\begin{aligned}\mathbf{F}(\mathbf{V}) &\approx \mathbf{F}(\mathbf{U}) + \mathbf{J}(\mathbf{V} - \mathbf{U}) \\ &= \mathbf{U} + \mathbf{J}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} = \mathbf{V} - \mathbf{U}\end{aligned}$$

  **J**: $J_{ki} = \partial f_k / \partial u_i$ is the **Jacobian** and $\boldsymbol{\epsilon}$ is the **absolute error**

$$\mathbf{V} = \mathbf{U} + \mathbf{J}\boldsymbol{\epsilon} + \beta \iff (\mathbf{I} - \mathbf{J})\,\boldsymbol{\epsilon} = \beta \iff \boldsymbol{\epsilon} = (\mathbf{I} - \mathbf{J})^{-1}\,\beta$$

$$\|\boldsymbol{\epsilon}\| \leq \|\,(\mathbf{I} - \mathbf{J})^{-1}\,\|\|\beta\|$$

- For the relative error:

$$\|\boldsymbol{\epsilon}_{\text{rel}}\| \leq \mathbf{M}\|\,(\mathbf{I} - \mathbf{J})^{-1}\,\|\epsilon_{\text{machine}}, \quad \mathbf{M} > 0.$$

# 1.9 condition and stability of functions

# condition of a function $f(x)$

**The Problem:**

Given a function

$$f : \mathbb{R}^m \to \mathbb{R}^k$$

compute the function value $f(x)$ for some $x \in \mathbb{R}^m$

**Definition:**

The *(relative) condition number* of a function is

$$\kappa(x) = \sup_{y \neq x} \frac{\|f(y) - f(x)\|/\|f(x)\|}{\|y - x\|/|x\|}$$

a local version is

$$\kappa(x) = \lim_{\epsilon \to 0} \sup_{\|y-x\|<\epsilon} \frac{\|f(y) - f(x)\|/\|f(x)\|}{\|y - x\|/|x\|}$$

or simplified $y = (1 + \epsilon S)x$ where $S$ is a diagonal matrix with $\pm 1$ diagonal elements

$$\kappa(x) = \limsup \frac{\|f((1 + \epsilon S)x) - f(x)\|}{}$$

# examples

1. $f(x) = 10x + 5$ (both global and local version are the same)

$$\kappa(x) = \sup_y \frac{10(x-y)/(10x+5)}{(x-y)/x}$$
$$= \frac{10x}{10x+5}$$

2. $f(x) = \sqrt{x}$ for $x > 0$

$$\kappa(f) = \sup_{y>0} \frac{(\sqrt{x}-\sqrt{y})/\sqrt{x}}{(x-y)/x}$$
$$= \sup_{y>0} \frac{\sqrt{x}}{\sqrt{x}+\sqrt{y}} = 1$$

- the local version is $\kappa(f) = 0.5$

# the difference $f(x_1, x_2) = x_1 - x_2$ can be ill-conditioned

$$\kappa(x) = \sup \frac{|x_1 - x_2 - y_1 + y_2| / |x_1 - x_2|}{\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \Big/ \sqrt{x_1^2 + x_2^2}}$$

- maximum obtained for $x_1 - y_1 = -(x_2 - y_2)$ and thus

$$\kappa(x) = \sqrt{2 \frac{x_1^2 + x_2^2}{(x_1 - x_2)^2}}$$

- condition number large for $x_1 \approx x_2$

# the exponential function

- $f(x) = \exp(x)$ for $x \in [0, M]$

$$\kappa(x) = \sup_{0 \le y \le M} \frac{|e^y - e^x|/e^x}{|y - x|/|x|} = \sup_y \frac{e^{y-x} - 1}{|y - x|} |x| < e^M |x|$$

  - as $|y - x| \le M$ and

$$\frac{e^{y-x} - 1}{y - x} = e^{\theta(y-x)}$$

    for some $\theta \in [0, 1]$ because the left hand side is the slope of a secant ...

- the local condition number is $\kappa(f) = |x|$

# condition number of a matrix

- matrix-vector product $f(x) = Ax$ for $x \in \mathbb{R}^n$

$$\kappa(A) = \sup \frac{\|A(x-y)\|/\|Ax\|}{\|x-y\|/\|x\|}$$
$$= \sup \frac{\|A(x-y)\|}{\|x-y\|} \cdot \frac{\|x\|}{\|Ax\|} = \|A\| \cdot \|A^{-1}\|$$

- it follows that $\kappa(A) = \kappa(A^{-1})$

# stability of numerical function $f(x, \delta)$

$$f : \mathbb{R}^m \otimes \mathbb{R}^k \to \mathbb{R}$$

models a function as evaluated on a computer

- where $\delta \in \mathbb{R}^k$ is an error parameter
- $f(x, 0)$ is the exact value

## Definition (stability)

$f(x, \delta)$ is *stable* if for any choice of

- $x \in \mathbb{R}^m$
- $\epsilon > 0$ and $\delta \in \mathbb{R}^k$ with $|\delta_k| \leq \epsilon$

there exist

- $y \in \mathbb{R}^m$ and $C_1, C_2 > 0$

such that $x$ is close to $y$, i.e.,

# a stronger and simpler condition

- concept used mostly in actual analysis

## Definition (backward stability)

$f(x, \delta)$ is *backward stable* if for any choice of

- $x \in \mathbb{R}^m$
- $\epsilon > 0$ and $\delta \in \mathbb{R}^k$ with $|\delta_k| \leq \epsilon$

there exist

- $y \in \mathbb{R}^m$
- $C > 0$

such that $x$ is close to $y$, i.e.,

$$\frac{\|y - x\|}{\|x\|} \leq C\epsilon$$

and $f(y, \delta)$ is equal to $f(y, 0)$

# accuracy of a backward stable algorithm

**Definition: relative error**

$$e = \frac{f(x,\delta) - f(x,0)}{|f(x,0)|}$$

**Proposition**

If $f(x,\delta)$ is backward stable and $f(x,0)$ is well conditioned with condition number $\kappa(x)$, then there is a $C > 0$ such that the relative error satisfies

$$|e| \leq \kappa(x)\, C\epsilon$$

for all rounding errors $\delta$ with $|\delta_k| \leq \epsilon$

*Proof.*

by backward stability and the definition of the condition number one has from backward stability some $y$ such that

$$\begin{aligned}\frac{|f(x,\delta)-f(x,0)|}{|f(x,0)|} &= \frac{|f(y,0)-f(x,0)|}{|f(x,0)|} \\ &\leq \kappa(x)\,\frac{\|y-x\|}{\|x\|} \\ &\leq C\kappa(x)\,\epsilon\end{aligned}$$

where $\|y-x\|/\|x\| \leq C\epsilon$

■

**Remarks**

- The constant $C$ depends on the algorithm and in particular the dimension of $\delta$
- Often it is easier to determine the constant $C$ and $\kappa$ then bounding the error directly

# example: $a - bc/d$ (Schur complement)

$$
\begin{aligned}
u_1 &= a \\
u_2 &= b \\
u_3 &= c \\
u_4 &= d \\
u_5 &= u_2 u_3 \\
u_6 &= u_5 / u_4 \\
u_7 &= u_1 - u_6
\end{aligned}
$$

- input $x = (a, b, c, d)$ (components of 2 by 2 matrix)
- Schur complement is major tool for Gaussian elimination
- backward stability has been used to get rounding error bounds for Gaussian elimination to differentiate between the effects of the algorithm and the effects of the data (the matrix)

# example: $a - bc/d$ with rounding errors

$$
\begin{aligned}
v_1 &= (1 + \delta_1)\, a \\
v_2 &= (1 + \delta_2)\, b \\
v_3 &= (1 + \delta_3)\, c \\
v_4 &= (1 + \delta_4)\, d \\
v_5 &= (1 + \delta_5)\, v_2 v_3 \\
v_6 &= (1 + \delta_6)\, v_5 / v_4 \\
v_7 &= (1 + \delta_7)\, (v_1 - v_6)
\end{aligned}
$$

# example: $a - bc/d$ backward stable model

$$
\begin{aligned}
z_1 &= (1 + \eta_1)\, a \\
z_2 &= (1 + \eta_2)\, b \\
z_3 &= (1 + \eta_3)\, c \\
z_4 &= (1 + \eta_4)\, d \\
z_5 &= z_2 z_3 \\
z_6 &= z_5 / z_4 \\
z_7 &= z_1 - z_6
\end{aligned}
$$

- the $\eta_k$ are a function of the $\delta_j$
- the result is the same as before $z_7 = v_7$

# example: $a - bc/d$ – compute the $\eta_j$

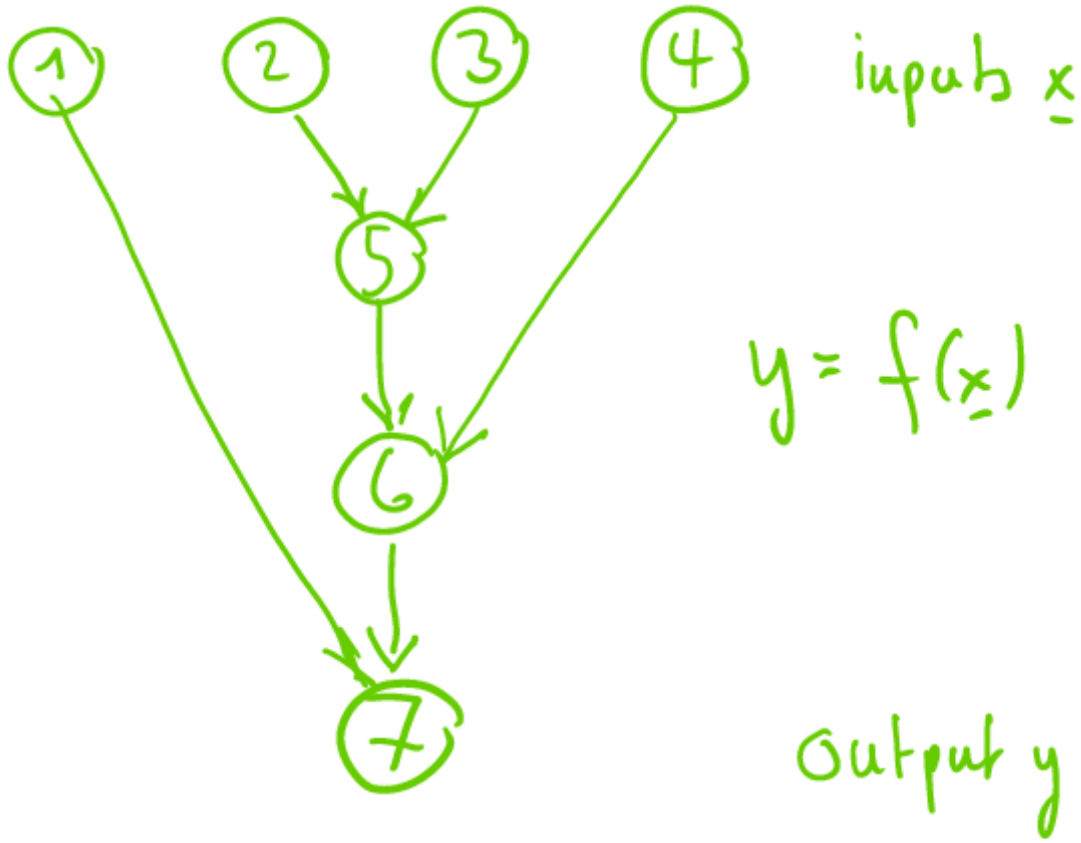
$$
\begin{aligned}
z_7 &= v_7 = (1+\delta_7)(v_1 - v_6) = z_1 - z_6 \\
z_6 &= (1+\delta_7)v_6 = (1+\delta_7)(1+\delta_6)v_5/v_4 = z_5/z_4 \\
z_5 &= (1+\delta_7)v_5 = (1+\delta_7)(1+\delta_5)v_2 v_3 = z_2 z_3 \\
z_4 &= (1+\delta_6)^{-1} v_4 = (1+\delta_6)^{-1}(1+\delta_4)d = (1+\eta_4)d \\
z_3 &= (1+\delta_7)v_3 = (1+\delta_7)(1+\delta_3)c = (1+\eta_3)c \\
z_2 &= (1+\delta_5)v_2 = (1+\delta_5)(1+\delta_2)b = (1+\eta_2)b \\
z_1 &= (1+\delta_7)v_1 = (1+\delta_7)(1+\delta_1)a = (1+\eta_1)a
\end{aligned}
$$

- thus one gets for the $\eta_j$

$$
\begin{aligned}
\eta_1 &= (1+\delta_7)(1+\delta_1) - 1 \\
\eta_2 &= (1+\delta_5)(1+\delta_2) - 1 \\
\eta_3 &= (1+\delta_7)(1+\delta_3) - 1 \\
\eta_4 &= (1+\delta_6)^{-1}(1+\delta_4) - 1
\end{aligned}
$$

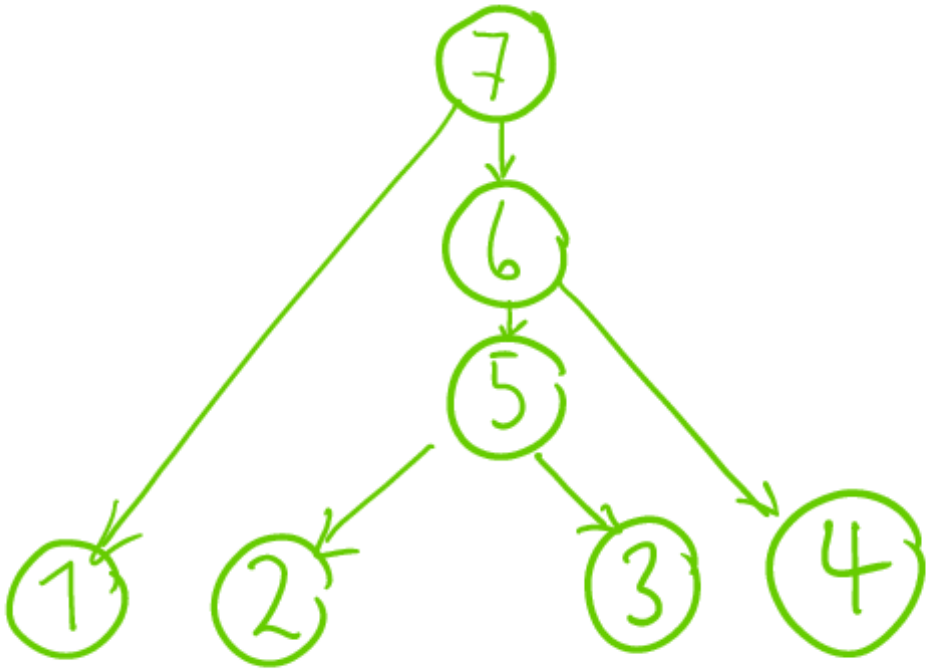
# Math3511 - graph of Schur complement

Wednesday, 7 March 2018 10:33 AM



# Math3511 - inverse graph

Wednesday, 7 March 2018 10:36 AM



# another example $f(x) = 1 + x$

- usual (global) error analysis from section 1.8

$$v_1 = (1 + \delta_1)x$$
$$v_2 = (1 + \delta_2)(1 + v_1)$$

- this gives for the result $v_2 = f(x, \delta)$ with $\delta = (\delta_1, \delta_2)$

$$v_2 = (1 + \delta_2)(1 + (1 + \delta_1)x) = (1 + \theta_2)(1 + x)$$

and from this one gets (neglecting small terms like $\delta_1\delta_2$ for the relative error $\theta_2$

$$\theta_2 = \frac{(1 + \delta_2)(1 + (1 + \delta_1)x)}{1 + x} - 1$$
$$\approx \frac{x}{1 + x}\delta_1 + \delta_2$$

thus the relative error is bounded by $(C + 1)\epsilon$ if $|x|/|1 + x| \leq C$ which we take as domain of $f$

# backward stability of $f(x, \delta)$ from previous slide

- $f(x, \delta)$ is backward stable if there is a $\zeta_1$ such that

$$f(x, \delta) = v_2 = z_2$$

for some $z_1, z_2$ and $\zeta_1$ with

$$z_1 = (1 + \zeta_1)x$$
$$z_2 = 1 + z_1$$

- solving backwards gives

$$1 + z_1 = z_2 = v_2 = 1 + (1 + \delta_2)v_1 + \delta_2$$

- and so

$$z_1 = (1 + \delta_2)v_1 + \delta_2 = (1 + \delta_2)(1 + \delta_1)x + \delta_2 = \zeta_1 x$$

and consequently

$$\zeta_1 = (1 + \delta_2)(1 + \delta_1) + \delta_2/x$$

- thus our "algorithm" $f(x, \delta)$ is backward stable if $|x| > 1/M > 0$

# condition number of $f(x) = 1 + x$

- the condition number of $f$ is

$$\kappa(f) = \sup_y \frac{|f(y) - f(x)|}{|y - x|} \frac{|x|}{|f(x)|}$$
$$= \frac{|x|}{|1 + x|}$$

- the condition number is large if $x \approx -1$ where the function is ill-conditioned but the function is well-conditioned otherwise