Librairies mathématiques et représentation des nombres en Python

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	**1**	**Librairies mathématiques et représentation des nombres en Python et erreurs liées**
Cours	2,3,4	Résolution des systèmes linéaires
Cours	5,6	Interpolation et Régression Linéaires
Cours	7	Zéro d'équation
Cours	8,9	Différentiation numérique
Cours	10	Intégration numérique
Cours	11,12	Introduction à l'optimisation
Cours	13	Rappel / Répétition

Outline

- **Python Reminders**
- **Scientific Computing with Numpy**
- **Plotting with Matplotlib**
- **Number representation**
- **Error & Error Propagation**

Python Reminders

Variables

- **Python type is dynamic**

```
b = 2 # b is integer type
print(b)
print(type(b))

b = b*2.0 # Now b is float type
print(b)
print(type(b))
```

executed in 9ms, finished 14:07:20 2019-09-03

```
2
<class 'int'>
4.0
<class 'float'>
```

Strings

A string is a sequence of characters enclosed in single or double quotes. Strings are *concatenated* with the plus (+) operator, whereas *slicing* (:) is used to extract a portion of the string. Here is an example:

```
>>> string1 = 'Press return to exit'
>>> string2 = 'the program'
>>> print(string1 + ' ' + string2)  # Concatenation
Press return to exit the program
>>> print(string1[0:12])            # Slicing
Press return
```

A string can be split into its component parts using the `split` command. The components appear as elements in a list. For example,

```
>>> s = '3 9 81'
>>> print(s.split())    # Delimiter is white space
['3', '9', '81']
```

- **Strings are immutable**

Tuples

A *tuple* is a sequence of *arbitrary objects* separated by commas and enclosed in parentheses. If the tuple contains a single object, a final comma is required; for example, `x = (2,)`. Tuples support the same operations as strings; they are also immutable. Here is an example where the tuple `rec` contains another tuple `(6,23,68)`:

```
>>> rec = ('Smith','John',(6,23,68))    # This is a tuple
>>> lastName,firstName,birthdate = rec  # Unpacking the tuple
>>> print(firstName)
John
>>> birthYear = birthdate[2]
>>> print(birthYear)
68
>>> name = rec[1] + ' ' + rec[0]
>>> print(name)
John Smith
>>> print(rec[0:2])
('Smith', 'John')
```

Lists

A list is similar to a tuple, but it is *mutable*, so that its elements and length can be changed. A list is identified by enclosing it in brackets. Here is a sampling of operations that can be performed on lists:

```
>>> a = [1.0, 2.0, 3.0]      # Create a list
>>> a.append(4.0)            # Append 4.0 to list
>>> print(a)
[1.0, 2.0, 3.0, 4.0]
>>> a.insert(0,0.0)          # Insert 0.0 in position 0
>>> print(a)
[0.0, 1.0, 2.0, 3.0, 4.0]
>>> print(len(a))            # Determine length of list
5
>>> a[2:4] = [1.0, 1.0, 1.0] # Modify selected elements
>>> print(a)
[0.0, 1.0, 1.0, 1.0, 1.0, 4.0]
```

Lists

If *a* is a mutable object, such as a list, the assignment statement `b = a` does not result in a new object *b*, but simply creates a new reference to *a*. Thus any changes made to *b* will be reflected in *a*. To create an independent copy of a list *a*, use the statement `c = a[:]`, as shown in the following example:

```
>>> a = [1.0, 2.0, 3.0]
>>> b = a                  # 'b' is an alias of 'a'
>>> b[0] = 5.0             # Change 'b'
>>> print(a)
[5.0, 2.0, 3.0]            # The change is reflected in 'a'
>>> c = a[:]               # 'c' is an independent copy of 'a'
>>> c[0] = 1.0             # Change 'c'
>>> print(a)
[5.0, 2.0, 3.0]            # 'a' is not affected by the change
```

Matrices (as lists)

Matrices can be represented as nested lists, with each row being an element of the list. Here is a 3×3 matrix a in the form of a list:

```
>>> a = [[1, 2, 3], \
         [4, 5, 6], \
         [7, 8, 9]]
>>> print(a[1])          # Print second row (element 1)
[4, 5, 6]
>>> print(a[1][2])       # Print third element of second row
6
```

The backslash (\) is Python's *continuation character*. Recall that Python sequences have zero offset, so that `a[0]` represents the first row, `a[1]` the second row, etc. With very few exceptions we do not use lists for numerical arrays. It is much more convenient to employ *array objects* provided by the `numpy` module. Array objects are discussed later.

Operators

- **Arithmetic Operators**

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation
%	Modular division

`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a*b`
`a /= b`	`a = a/b`
`a **= b`	`a = a**b`
`a %= b`	`a = a%b`

- **Comparison Operators**

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Conditions

```
if condition:
    block
```

```
elif condition:
    block
```

```
else:
    block
```

```
def sign_of_a(a):
    if a < 0.0:
        sign = 'negative'
    elif a > 0.0:
        sign = 'positive'
    else:
        sign = 'zero'
    return sign
```

Loops

- **While Loops**

```
while condition:
    block
```

```
else:
    block
```

```
nMax = 5
n = 1
a = []                  # Create empty list
while n < nMax:
    a.append(1.0/n)     # Append element to list
    n = n + 1
print(a)
```

- **For Loops**

```
for target in sequence:
    block
```

```
break
```

```
continue
```

```
x = []                      # Create an empty list
for i in range(1,100):
  if i%7 != 0: continue     # If not divisible by 7, skip rest of loop
  x.append(i)               # Append i to the list
print(x)
```

The printout from the program is

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

Loops

- **While Loops**

```
while condition:
    block
```

```
else:
    block
```

```
nMax = 5
n = 1
a = []                  # Create empty list
while n < nMax:
    a.append(1.0/n)     # Append element to list
    n = n + 1
print(a)
```

- **For Loops**

```
for target in sequence:
    block
```

```
break
```

```
continue
```

```
x = []                      # Create an empty list
for i in range(1,100):
  if i%7 != 0: continue     # If not divisible by 7, skip rest of loop
  x.append(i)               # Append i to the list
print(x)
```

The printout from the program is

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

Miscellaneous

- **Type Conversion**

Type conversions can also achieved by the following functions:

`int(a)`	Converts *a* to integer
`float(a)`	Converts *a* to floating point
`complex(a)`	Converts to complex $a + 0j$
`complex(a,b)`	Converts to complex $a + bj$

These functions also work for converting strings to numbers as long as the literal in the string represents a valid number. Conversion from a float to an integer is carried out by truncation, not by rounding off. Here are a few examples:

- **Mathematical Functions**

Core Python supports only the following mathematical functions:

`abs(a)`	Absolute value of `a`
`max(`*sequence*`)`	Largest element of *sequence*
`min(`*sequence*`)`	Smallest element of *sequence*
`round(a,n)`	Round `a` to `n` decimal places
`cmp(a,b)`	Returns $\begin{cases} -1 \text{ if } a < b \\ 0 \text{ if } a = b \\ 1 \text{ if } a > b \end{cases}$

The majority of mathematical functions are available in the `math` module.

- **Error Control**

```
try:
    do something
except error:
    do something else
```

Miscellaneous

- **Function**

```
def func_name(param1, param2,...):
    statements
    return return_values
```

- **Lambda Function**

If the function has the form of an expression, it can be defined with the lambda statement

```
func_name = lambda param1, param2,...: expression
```

Multiple statements are not allowed.

Here is an example:

```
>>> c = lambda x,y : x**2 + y**2
>>> print(c(3,4))
25
```

- **Modules**

It is sound practice to store useful functions in modules. A module is simply a file where the functions reside; the name of the module is the name of the file. A module can be loaded into a program by the statement

```
from module_name import *
```

Math Module or CMath

- **Import**

```
from math import *
```

```
from math import func1, func2, ...
```

```
import math
```

```
import math as m
```

```
>>> from math import log,sin
>>> print(log(sin(0.5)))
-0.735166686385
```

```
>>> import math as m
>>> print(m.log(m.sin(0.5)))
-0.735166686385
```

- **List Content of the module**

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan',
 'atan2', 'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs',
 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
 'log10', 'modf', 'pi', 'pow', sign', sin', 'sinh',
 'sqrt', 'tan', 'tanh']
```

- **Cmath for “complex” numbers**

```
['__doc__', '__name__', 'acos', 'acosh', 'asin', 'asinh',
 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'log',
 'log10', 'pi', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Numpy

Numpy

- **Numpy**
  - Numpy is the core library for scientific computing in Python.
  - It provides a high-performance multidimensional array object, and tools for working with these arrays.
  - Generally much faster than python “loops” because
    - It is backend by C++ functions
    - It uses CPU vectorization wherever it’s possible
- Import Convention:

```
import numpy as np
```

  - Not always followed in the slides, because the reference book doesn’t always use it

Arrays creation

- **From List:**

`array(list, type)`

```
>>> from numpy import array
>>> a = array([[2.0, -1.0],[-1.0, 3.0]])
>>> print(a)
[[ 2. -1.]
 [-1.  3.]]
```

- **Array of Zeros:**

`zeros((dim1, dim2), type)`

- **Array of Ones:**

`ones((dim1, dim2), type)`

- **Array from Range:**

`arange(from, to, increment)`

Accessing and Changing array elements

If *a* is a rank-2 array, then `a[i,j]` accesses the element in row *i* and column *j*, whereas `a[i]` refers to row *i*. The elements of an array can be changed by assignment as follows:

```
>>> from numpy import *
>>> a = zeros((3,3),int)
>>> print(a)
[[0 0 0]
 [0 0 0]
 [0 0 0]]
>>> a[0] = [2,3,2]      # Change a row
>>> a[1,1] = 5          # Change an element
>>> a[2,0:2] = [8,-3]   # Change part of a row
>>> print(a)
[[ 2  3  2]
 [ 0  5  0]
 [ 8 -3  0]]
```

Operations on arrays

- **Broadcasting !**

Arithmetic operators work differently on arrays than they do on tuples and lists—the operation is *broadcast* to all the elements of the array; that is, the operation is applied to each element in the array. Here are examples:

```
>>> from numpy import array
>>> a = array([0.0, 4.0, 9.0, 16.0])
>>> print(a/16.0)
[ 0.      0.25    0.5625  1.    ]
>>> print(a - 4.0)
[ -4.   0.   5.  12.]
```

The mathematical functions available in numpy are also broadcast, as follows:

```
>>> from numpy import array,sqrt,sin
>>> a = array([1.0, 4.0, 9.0, 16.0])
>>> print(sqrt(a))
[ 1.  2.  3.  4.]
>>> print(sin(a))
[ 0.84147098 -0.7568025   0.41211849 -0.28790332]
```

Array functions

```
>>> from numpy import *
>>> A = array([[4,-2,1],[-2,4,-2],[1,-2,3]],float)
>>> b = array([1,4,3],float)
>>> print(diagonal(A))        # Principal diagonal
[ 4.  4.  3.]
>>> print(diagonal(A,1))      # First subdiagonal
[-2. -2.]
>>> print(trace(A))           # Sum of diagonal elements
11.0
>>> print(argmax(b))          # Index of largest element
1
>>> print(argmin(A,axis=0))   # Indices of smallest col. elements
[1 0 1]

>>> print(identity(3))        # Identity matrix
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Array multiplications

```
from numpy import *
x = array([7,3])
y = array([2,1])
A = array([[1,2],[3,2]])
B = array([[1,1],[2,2]])

# Dot product
print("dot(x,y) =\n",dot(x,y))       # {x}.{y}
print("dot(A,x) =\n",dot(A,x))       # [A]{x}
print("dot(A,B) =\n",dot(A,B))       # [A][B]

# Inner product
print("inner(x,y) =\n",inner(x,y))   # {x}.{y}
print("inner(A,x) =\n",inner(A,x))   # [A]{x}
print("inner(A,B) =\n",inner(A,B))   # [A][B_transpose]

# Outer product
print("outer(x,y) =\n",outer(x,y))
print("outer(A,x) =\n",outer(A,x))
print("outer(A,B) =\n",outer(A,B))
```

- 17
- [13 27]
- [[5 5]
  [7 7]]

- 17
- [13 27]
- [[3 6]
  [5 10]]

- [[14 7]
  [6 3]]
- [[7 3]
  [14 6]
  [21 9]
  [14 6]]
- [[1 1 2 2]
  [2 2 4 4]
  [3 3 6 6]
  [2 2 4 4]]

Linear Algebra Module

The numpy module comes with a linear algebra module called `linalg` that contains routine tasks such as matrix inversion and solution of simultaneous equations. For example,

```
>>> from numpy import array
>>> from numpy.linalg import inv,solve
>>> A = array([[ 4.0, -2.0,  1.0], \
               [-2.0,  4.0, -2.0], \
               [ 1.0, -2.0,  3.0]])
>>> b = array([1.0, 4.0, 2.0])
>>> print(inv(A))                          # Matrix inverse
[[ 0.33333333  0.16666667  0.        ]
 [ 0.16666667  0.45833333  0.25      ]
 [ 0.          0.25        0.5       ]]
>>> print(solve(A,b))                      # Solve [A]{x} = {b}
[ 1. ,  2.5,  2. ]
```

We will re-implement many of these functions during this course,
but we can verify our solutions with the default numpy implementation

Miscellaneous

Copying Arrays

We explained earlier that if *a* is a mutable object, such as a list, the assignment statement `b = a` does not result in a new object *b*, but simply creates a new reference to *a*, called a *deep copy*. This also applies to arrays. To make an independent copy of an array *a*, use the `copy` method in the `numpy` module:

```
b = a.copy()
```

Vectorizing Algorithms

Sometimes the broadcasting properties of the mathematical functions in the `numpy` module can be used to replace loops in the code. This procedure is known as vectorization. Consider, for example, the expression

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} \sin \frac{i\pi}{100}$$

Python version

```
from math import sqrt,sin,pi
x = 0.0; s = 0.0
for i in range(101):
    s = s + sqrt(x)*sin(x)
    x = x + 0.01*pi
print(s)
```

Executed in 241µs

Numpy (vectorized) version

```
from numpy import sqrt,sin,arange
from math import pi
x = arange(0.0, 1.001*pi, 0.01*pi)
print(sum(sqrt(x)*sin(x)))
```

Executed in 9µs

The vectorized algorithm executes much faster, but uses more memory

Plotting with Matplotlib

Matplotlib

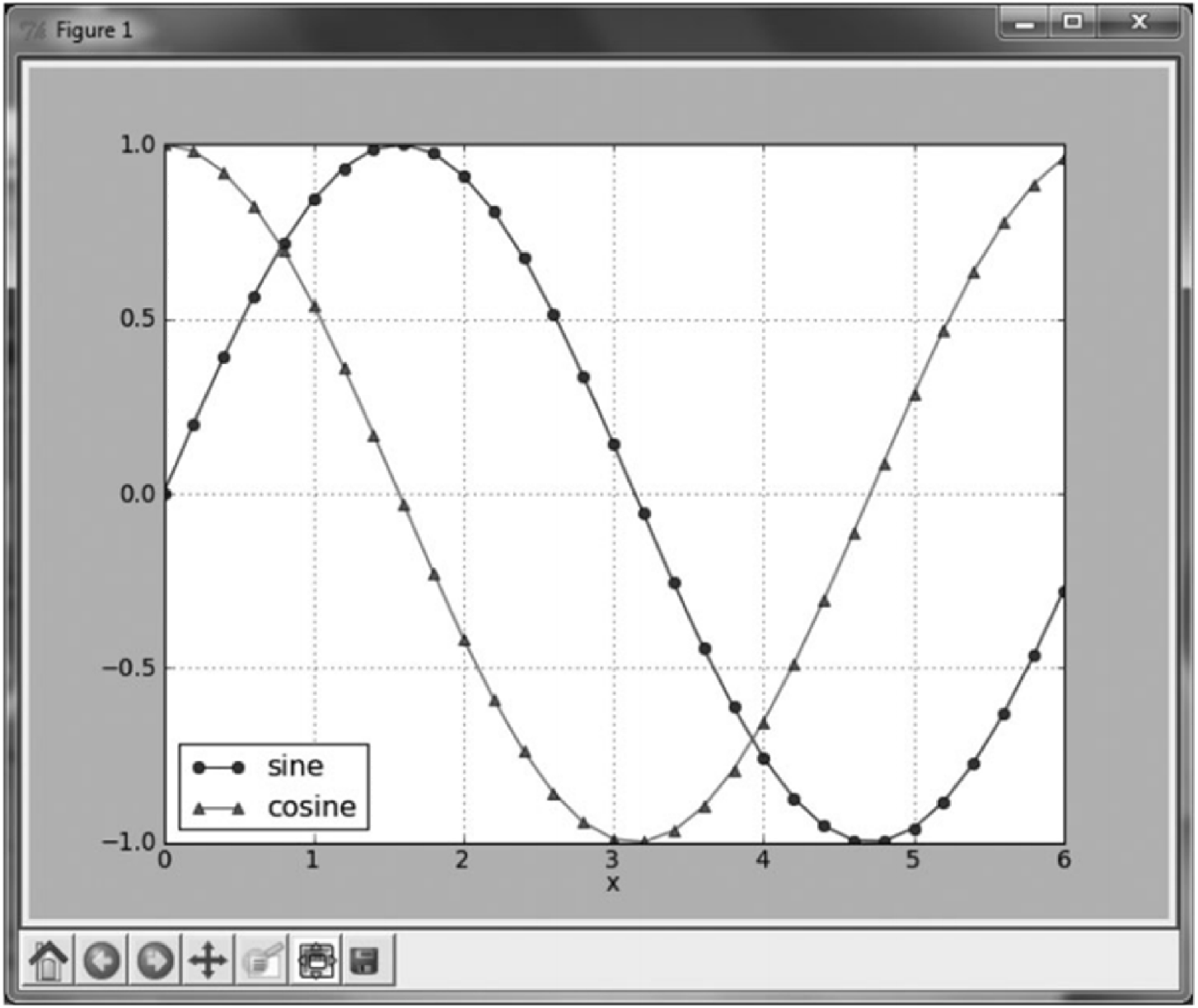
- **Matplotlib**
  - Matplotlib is a Python 2D plotting library
  - Matplotlib can be used in Python scripts, the Python and IPython shells, Jupyter…

- **Example**

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)

plt.plot(x,sin(x),'o-',x,cos(x),'^-')    # Plot with specified
                                         # line and marker style
plt.xlabel('x')                          # Add label to x-axis
plt.legend(('sine','cosine'),loc = 0)    # Add legend in loc. 3
plt.grid(True)                           # Add coordinate grid
plt.savefig('testplot.png',format='png') # Save plot in png
                                         # format for future use
plt.show()                               # Show plot on screen
input("\nPress return to exit")
```

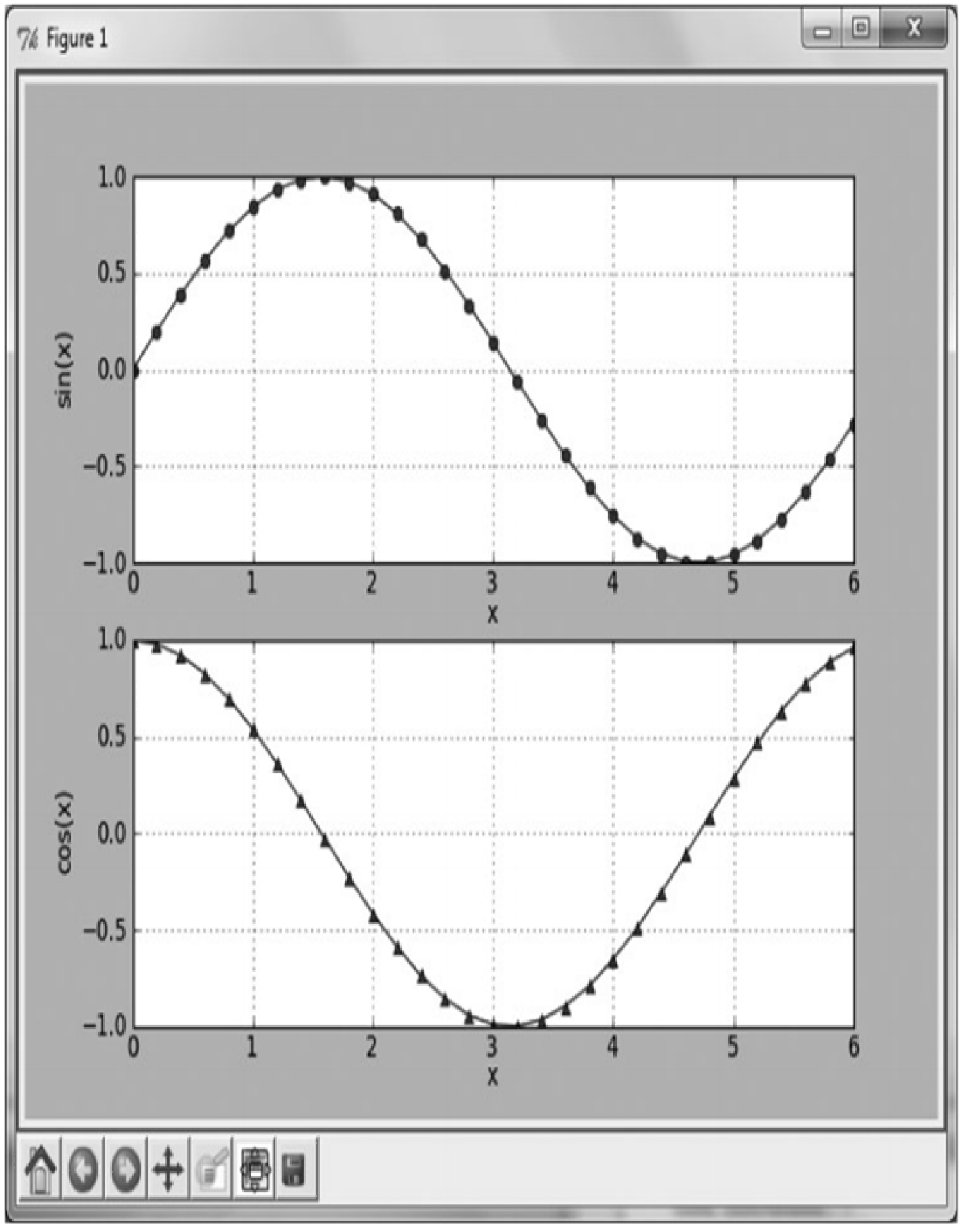
Example



Multiple plot with subplot

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,6.2,0.2)

plt.subplot(2,1,1)
plt.plot(x,sin(x),'o-')
plt.xlabel('x');plt.ylabel('sin(x)')
plt.grid(True)
plt.subplot(2,1,2)
plt.plot(x,cos(x),'^-')
plt.xlabel('x');plt.ylabel('cos(x)')
plt.grid(True)
plt.show()
input("\nPress return to exit")
```



Miscellaneous

Line & Marker Style

'-'	Solid line
'--'	Dashed line
'-.'	Dash-dot line
':'	Dotted line
'o'	Circle marker
'^'	Triangle marker
's'	Square marker
'h'	Hexagon marker
'x'	x marker

Color Strings

Alias	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

Legend Location

0	"Best" location
1	Upper right
2	Upper left
3	Lower left
4	Lower right

Color as hex string RGB (as in html)

```
color = '#eeefff'
```

Number Representation

Why is it important ?

```
print(53.0/10.0 == 53.0*0.1)
```

False

```
x=1.0;
x-=0.1; print( x == 0.9 );
x-=0.1; print( x == 0.8 );
x-=0.1; print( x == 0.7 );
```

True
True
False

```
z=0.00000000000000001;
print( z == 1e-17 );
print( z*10.0 == 1e-16 );
print( z/10.0 == 1e-18 );
z+=1.0;
z-=1.0;
print( z/10.0 == 1e-18 );
```

True
False
True

False

Numbers in Numerical Algorithms

- Most numerical algorithms work with numbers from $\mathbb{R}$
- In most programming languages there are special primitive datatypes for such numbers

Numpy type	C type	Description
np.int8	`int8_t`	Byte (-128 to 127)
np.int16	`int16_t`	Integer (-32768 to 32767)
np.int32	`int32_t`	Integer (-2147483648 to 2147483647)
np.int64	`int64_t`	Integer (-9223372036854775808 to 9223372036854775807)
np.uint8	`uint8_t`	Unsigned integer (0 to 255)
np.uint16	`uint16_t`	Unsigned integer (0 to 65535)
np.uint32	`uint32_t`	Unsigned integer (0 to 4294967295)
np.uint64	`uint64_t`	Unsigned integer (0 to 18446744073709551615)
np.intp	`intptr_t`	Integer used for indexing, typically the same as `ssize_t`
np.uintp	`uintptr_t`	Integer large enough to hold a pointer
np.float32	`float`	
*np.float64 / np.float_*	`double`	Note that this matches the precision of the builtin python *float*.
np.complex64	`float complex`	Complex number, represented by two 32-bit floats (real and imaginary components)
*np.complex128 / np.complex_*	`double complex`	Note that this matches the precision of the builtin python *complex*.

- How do they work?

Representation of (Integer) Numbers

- Most languages use the machine representation used by the CPU

- Integral types:
  - In C you also have signed and unsigned integers.
    - Size of type depends on machine!
    - char (at least 8 bits), short (at least 16 bits), int (at least 32bits), long, ...
    - Signed as two's completement integers
  - In Python:
    - Only Signed integer
    - Only int (32 bit or 64 bit depending on machine)

Example: unsigned char type in C

- 8-bit integer $[0, 2^8 - 1]$

- Example: 222

$$222 = 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$$

- Binary (base-2) representation in one byte:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	1	1	0

Example: signed char in C

- 8-bit two's complement integer $[-2^7, 2^7 - 1]$
- Negative values represented as ~|x|+1
- Example: $34 = 2^5 + 2^1$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	0	0	1	0

- Example: -34
  - $|-34| = 34 = 2^5 + 2^1$
  - $\sim 34 = 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^0$
  - $\sim 34 + 1$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	1	1	0

Fixed-Point Representation

- A real number x is represented by the integer $2^k \cdot x$

- Example: $k = 8$
  $x = 2.5$ is represented by integer $256 \cdot 2.5 = 640$

- Addition of two numbers x and y:
  $x + y$ represented by integer:$(2^k \cdot x) + (2^k \cdot y)$

- Multiplication of two numbers x and y:
  $x \cdot y$ represented by integer: $(2^k \cdot x) \cdot (2^k \cdot y) / \, 2^k$

Fixed-Point Representation

- Why factor 2^k?
- Very convenient representation in hardware
- If we choose $k = 8$ and 16-bit integers we get:

$$2.5 \rightarrow \quad 2^8 \cdot 2.5 = 640$$

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
Integral part = 2								Non-integral part = 0.5							

- But: not all numbers can be represented!
  - The decimal part is represented in steps of $\frac{1}{2^k}$
  - For $k = 8,\ 2.6$ is approximated by 2.6015625

Fixed-Point Representation

- Very easy to implement (bit shifting!)
- But: How many bits would you need for a variable that can hold very large numbers (3,000,000,000) *and* very small numbers (0.0000000003) ?
  - Many bits for the integral part + many bits for the non-integral part
  - What a waste of space... Both numbers have only one “significant” digit 3!

Floating-Point Representation

- Idea: Separate the *significand* and the *exponent*

$$3{,}000{,}000{,}000 \rightarrow 3 \cdot 10^{9}$$

$$0.0000000003 \rightarrow 3 \cdot 10^{-9}$$

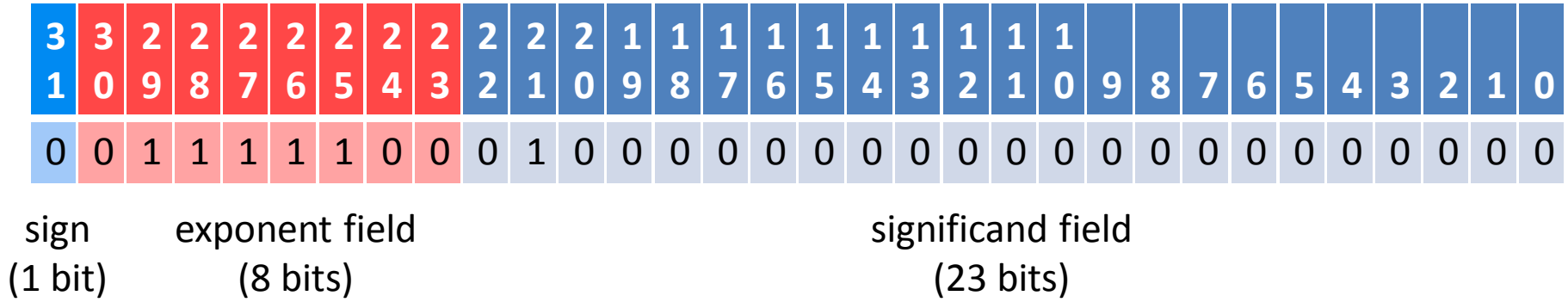
and store these two numbers in a binary format.

- In the following, we will see the *IEEE 754 standard* supported by most modern computers.
- IEEE 754 describes the format and also how operations should behave (rounding, division by zero,…)
- Operations with IEEE 754 floating-point numbers are performed in software (libraries) or in hardware (CPU)

IEEE 754

- The two most popular formats are
  - “binary32” (“single precision”): 32-bit, C’s “float” type
  - “binary64” (“double precision”): 64-bit, C’s “double” type
- In C, format of “float” and “double” are implementation specific!
- C also has a “long double” format, often 80-bit format (not defined by IEEE 754)

IEEE 754 Single Precision (32 bit)



- $Value = (-1)^{b_{31}} \cdot (\mathbf{1}.b_{22} \dots b_0)_2 \cdot 2^{(b_{30} \dots b_{23})_2 - 127}$
- The number is always stored *normalized* (leading **1** bit)
- Example (from Wikipedia):
  - Sign bit is 0 $\rightarrow$ positive number
  - Significand = $(\mathbf{1}.01000000000000000000000)_2 = 1 + 2^{-2}$
  - Exponent (with base 2) = $(01111100)_2 - 127 = -3$
  - Result: $1.25 \cdot 2^{-3} = 0.15625$

IEEE 754 Single Precision (32 bit)

- Another example:

Decimal $= -13.625$

$= -(2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-3})$

Binary $= -(1101.101)_2$

Normalized $= -(\mathbf{1}.101101)_2 \cdot 2^3$

Sign bit = 1

Significand field = 10110100000000000000000

Exponent field = 10000010 (= 3+127)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IEEE 754 Single Precision

- Exponent field $(b_{30} \ldots b_{23})_2$ has special meanings:
  - 0

    $Value = (-1)^{b_{31}} \cdot (\mathbf{0}.b_{22} \ldots b_0)_2 \cdot 2^{-126}$

    (used to represent very small numbers)
  - 1 .. 254

    $Value = (-1)^{b_{31}} \cdot (\mathbf{1}.b_{22} \ldots b_0)_2 \cdot 2^{(b_{30} \ldots b_{23})_2 - 127}$

    (used for normal numbers)
  - 255

    $$Value = \begin{cases} \pm\infty, & if\ significant\ bits\ are\ 0 \\ NaN, & if\ significant\ bits\ are\ \neq 0 \end{cases}$$

NaN and Infinity

- Operations return *NaN* (Not a Number) when
  - Result is indeterminate, for example $\frac{0}{0}$ or $0 \cdot \infty$.
  - Result is complex, for example $\sqrt{-1}$
  - Operation with another NaN, for example 3+ $\frac{0}{0}$

- Operations return $+\infty$ or $-\infty$ when
  - Maximum is exceeded (overflow), for example $10^{308} \cdot 10$
  - Non-zero value divided by 0

IEEE 754 Single and Double Precision

	Single (32 bit)	Double (64 bit)
Significand field	23 bits	52 bits
Exponent field	8 bits	11 bits
Exponent bias	127	1023
Min value	±1.40129846432481707e-45	±4.94065645841246544e-324
Max value	±3.40282346638528860e+38	±1.79769313486231570e+308

Consequences (1)

- Representable range is finite!
  Overflow will return ∞

- Therefore

(1e200 * 1e150) * 1e-100 ≠ 1e200 * (1e150 * 1e-100)

→ Order of operations is important!

Consequences (1): Example

Compute norm of vector x $= \sqrt{\sum_i x_i^2}$

```
from math import sqrt
def norm(x):
    squaredSum = 0;
    for x_i in x:
        squaredSum += x_i*x_i
    return sqrt(squaredSum)
```

Depending on the size of x, the variable squaredSum might overflow although the norm of x would fit in a double-precision variable!

```
print("%e" % norm([1e100, 1e100]))
print("%e" % norm([1e200, 1e200]))
```

executed in 11ms, finished 13:47:36 2019-09-04

```
1.414214e+100
inf
```

Consequences (1): Improved example

 # Compute norm of vector x = $\sqrt{\sum_i x_i^2}$

 # Scale down each element to reduce the risk of overflow

```
def norm_improved(x):
    maxElement = max(x)
    squaredSum = 0
    for x_i in x:
        #scale x_i
        x_i_scaled = x_i/maxElement
        squaredSum += x_i_scaled*x_i_scaled

    #don't forget to unscale the result
    return sqrt(squaredSum) * maxElement
```

There is no overflow anymore

```
print("%e" % norm_improved([1e100, 1e100]))
print("%e" % norm_improved([1e200, 1e200]))
```

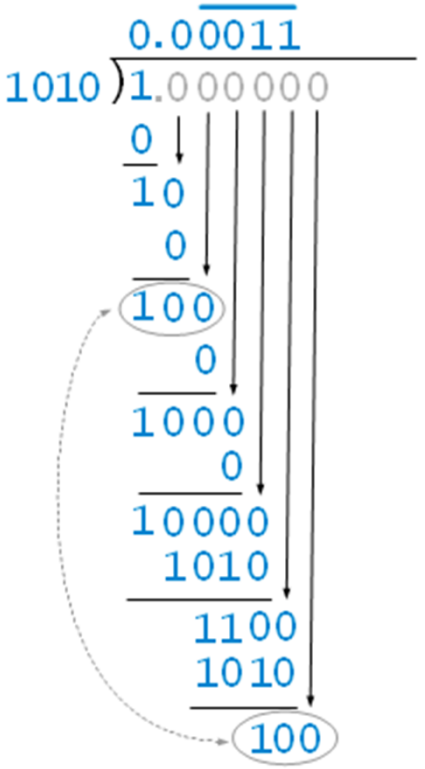
executed in 11ms, finished 13:50:07 2019-09-04

```
1.414214e+100
1.414214e+200
```

Consequences (2)

- Some numbers cannot be represented in base 2 with a finite significand:

$$0.1 = (0.0\overline{0011}_2)$$
$$= (0.000^{1}\ 00110^{0}\ \ \ldots)_2$$
$$= (\frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \cdots)_{10}$$
$$= (0.062^{5} + 0.03125 + 0.00390625 + \ldots)_{10}$$



- Therefore

$$53.0 / 10.0 \neq 53.0 * 0.1$$

Consequences (3)

- Precision is finite!
  - Non-representable results will be rounded
  - Even if the two numbers
    1 and 2^{-30} can be represented in single-precision,
    $(1 + 2^{-30})$ cannot be represented without loss!

  - Never compare floating-point numbers directly!

    Instead of if (a == b)

    do if (Math.abs(a-b) < EPSILON)

    (with EPSILON=1e-20 for example)

Consequences (3): Example

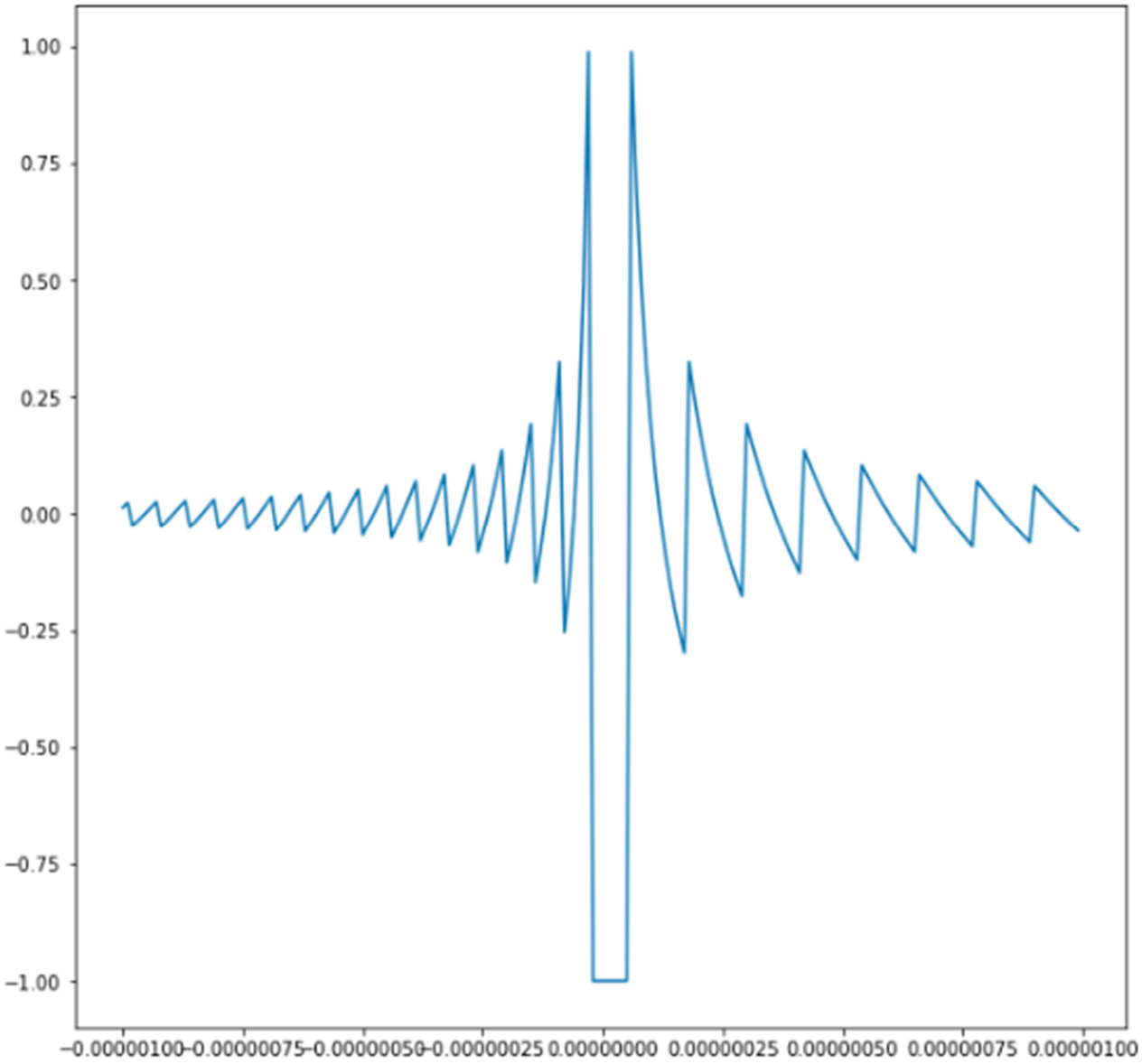
- Consider this function:

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

- In theory, we have $f(x) \to 0$ for $x \to 0$

- In practice, even small differences from the theoretical result caused by rounding errors will “blow up” because of the division by x

Consequences (3): Example



Errors and Error Propagation

Errors

- Numerical algorithms generally provides approximated solution because numerical errors are often introduced…

- x → real value
- $\tilde{x}$ → approximated value = $x + e_x$

- The error itself is unknown
  - otherwise we would correct it
  - but it is often small and bounded ($|e| < e_{max}$)

- Absolute Error: $e_x = \tilde{x} - x$
- Relative Error: $\varepsilon_x = \frac{\tilde{x} - x}{x}$

Sources of Errors in Numerical Calculus

- Some sources of errors are:
  - Intrinsic data errors
  - Discretization errors
  - Convergence errors
  - Rounding errors
  - …

Truncation errors

Intrinsic errors

- Input data of the program generally already have errors on it
  - Error from the instrument that took the measure
  - Error from anterior algorithm that produced this data
  - Error from the human who badly wrote the value
  - Etc…

- There is nothing we can do about these errors,
  - but we should make sure that our algorithms are robust against small errors (see ill-conditioning)

$$f(x) \rightarrow \varepsilon_f \approx \varepsilon_x$$

  - The error on the input (x) should not be amplified by our algorithm (f)

Discretization Errors

- Because the computer is a “finite” system
  - **Continuous problems** are replaced by **discrete problems**

$$\int_a^b f(x)dx \cong \sum_{i=1}^{n} k_i f(x_i)$$

- The function is evaluated in a **finite** (n) number of point

  → Discretization errors (larger for small n)

- This error is independent from rounding errors that are caused by the function evaluation itself.

Convergence errors

- Many numerical algorithms are solving problems by iteratively getting closer to the solution

- They provide an exact solution only to the limit of the series
  → Infinite number of iterations
  → Not possible to run the algorithm for ever,
  so the iteration are generally stop when the residual error is small
  → Convergence errors

- Example:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right); \quad (\mathrm{n} = 0,1,2, \dots)$$

$$\lim_{n\to\infty} x_n = \sqrt{2}\,; \qquad (\text{avec } x_0 = 1)$$

Step	Value	Convergence Error
x_0	1	$4\ 10^{-1}$
x_1	1.5	$8\ 10^{-2}$
x_2	1.416666666	$2\ 10^{-3}$
x_3	1.414215686	$2\ 10^{-6}$

We should decide when to stop the iterations to make this error small in comparison with rounding errors

Rounding Errors

- Rounding errors comes from the truncation of numbers to a limited number of cyphers
  - This is true for both the decimal and binary encoding

- **π** = 3.14159265 is an approximation with p=9 cyphers
  - The error is of the order of $5\ 10^{-9}$ **$= 10^{-(p-1)} / 2$**

- Floating numbers

	Single (32 bit)	Double (64 bit)
Significand field	23 bits → **p=24**	52 bits → **p=53**
Exponent field	8 bits	11 bits
Exponent bias	127	1023
Min value	±1.40129846432481707e-45	±4.94065645841246544e-324
Max value	±3.40282346638528860e+38	±1.79769313486231570e+308
Epsilon Machine ε_x	$2^{-24} \approx$ 5.96e-08	$2^{-53} \approx$ 1.11e-16

Thanks to normalized encoding, we get one more bit for free

Rounding Error on float

- Real number x is approximated by its float representation
  X → fl(x)

- $e = |\tilde{x} - x| = |fl(x) - x| \leq 2^{-p}\, 2^{t}$
  – with t the exponent in float representation

- $\varepsilon = \frac{|\tilde{x}-x|}{|x|} \leq |fl(x) - x| \leq 2^{-p}$

- $fl(x) = x(1 + \varepsilon)$ with $|\varepsilon| \leq 2^{-p}$

Multiplications of floats

- Operations on floats:

$x \oplus y$	fl(x + y)
$x \ominus y$	fl(x - y)
$x \otimes y$	fl(x × y)
$x \oslash y$	fl(x / y)

- Multiplication

$$
\begin{aligned}
fl(x) \otimes fl(y) &= fl\big(fl(x) \times fl(y)\big) \\
&= \big(fl(x) \times fl(y)\big)\,(1+\varepsilon) \\
&= \big(x(1+\varepsilon') \times y(1+\varepsilon'')\big)\,(1+\varepsilon) \\
&= (x \times y)(1+\varepsilon')\,(1+\varepsilon'')(1+\varepsilon) \\
&\simeq (x \times y)(1+|\varepsilon+\varepsilon'+\varepsilon''|)
\end{aligned}
$$

The error relative on float multiplication is around $|\varepsilon+\varepsilon'+\varepsilon''|$

Subtraction of floats

- Subtraction

$$
\begin{aligned}
fl(x) \ominus fl(y) &= fl\big(fl(x) - fl(y)\big) \\
&= \big(fl(x) - fl(y)\big)\,(1+\varepsilon) \\
&= (x(1+\varepsilon') - y(1+\varepsilon''))\,(1+\varepsilon) \\
&= (x-y)\left[1 + \varepsilon' \frac{x}{x-y} - \varepsilon'' \frac{y}{x-y}\right](1+\varepsilon)
\end{aligned}
$$

The error relative on float subtraction is around $\left|\varepsilon + \left[\varepsilon' \frac{x}{x-y} - \varepsilon'' \frac{x}{x-y}\right](1+\varepsilon)\right|$

If x and y are close to each other
→ (x-y) is small
→ 1/(x-y) get very large

→ There might be a loss of signification on $fl(x) \ominus fl(y)$

Error propagation of generic functions

- f a function with n variables $x_0, x_1, \ldots, x_n$

$$e_i = \tilde{x}_i - x_i$$

$$e_f = f(\tilde{x}_0, \ldots, \tilde{x}_n) - f(x_0, \ldots, x_n)$$

$$e_f \simeq \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x_0, \ldots, x_n)\, e_i$$

$$e_f \lesssim \sum_{i=1}^{n} \left|\frac{\partial f}{\partial x_i}(x_0, \ldots, x_n)\right| |e_i|$$

- Si $f(x_0, \ldots, x_n) = x_0 \ldots x_n$ (**multiplication**)

$$e_f \approx \sum_{i=1}^{n} \frac{1}{x_i} f(x_0, \ldots, x_n)\, e_i \qquad \rightarrow \qquad \varepsilon_f \approx \sum_{i=1}^{n} \varepsilon_i$$

Sum of relative errors for product

Error propagation (quotient & addition)

- Si $f(x_1, x_2) = x_1/x_2$ (quotient)

$$e_f \approx \frac{1}{x_2} e_1 + \frac{(-x_1)}{{x_2}^2} e_2 \qquad \rightarrow \qquad \varepsilon_f \approx \varepsilon_1 - \varepsilon_2$$

→ Difference of relative errors for quotient

- Si $f(x_0, \ldots, x_n) = x_0 + \cdots + x_n$ (addition)

$$e_f \approx \sum_{i=1}^{n} 1 \; e_i \qquad \rightarrow \qquad \varepsilon_f \approx \sum_{i=1}^{n} \varepsilon_i \; \frac{x_i}{x_0 + \cdots + x_n}$$

→ Sum of relative errors multiplied by the factor $x_i/(x_0 + \cdots + x_n)$
Small (<1) if all terms are of same sign… but could be large if the sum contains different signs (positive and negative terms)

Error propagation (quotient & addition)

- Si $f(x_1, x_2) = x_1/x_2$ (quotient)

$$e_f \approx \frac{1}{x_2} e_1 + \frac{(-x_1)}{{x_2}^2} e_2 \qquad \rightarrow \qquad \varepsilon_f \approx \varepsilon_1 - \varepsilon_2$$

→ Difference of relative errors for quotient

- Si $f(x_0, \ldots, x_n) = x_0 + \cdots + x_n$ (addition)

$$e_f \approx \sum_{i=1}^{n} 1 \; e_i \qquad \rightarrow \qquad \varepsilon_f \approx \sum_{i=1}^{n} \varepsilon_i \; \frac{x_i}{x_0 + \cdots + x_n}$$

→ Sum of relative errors multiplied by the factor $x_i/(x_0 + \cdots + x_n)$
Small (<1) if all terms are of same sign… but could be large if the sum contains different signs (positive and negative terms)