Ch.6: Array computing and curve plotting (Part 1)

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Sep 14, 2020

0.1 Plan for week 38

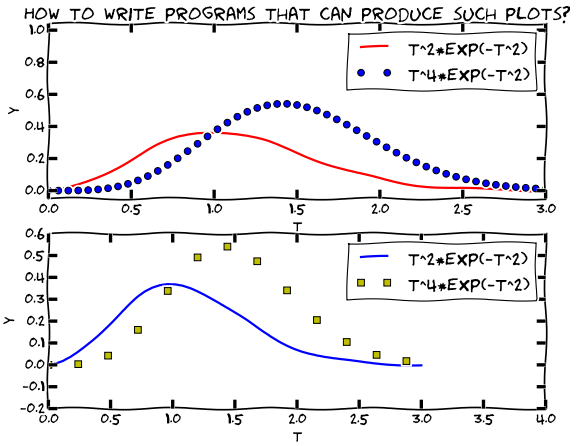
Monday 14 september

- Live programming of ex 4.4, 4.5, 4.13
- Intro to NumPy arrays and plotting
- Ex 5.7, 5.9

Wednesday 16 september

- Live programming of ex 5.10, 5.11, 5.13
- Plotting with `matplotlib`
- Making movies and animations from plots

0.2 Goal: learn to visualize functions



0.3 We need to learn about a new object: array

- Curves $y = f(x)$ are visualized by drawing straight lines between points along the curve
- Need to store the coordinates of the points along the curve in lists or *arrays* `x` and `y`
- Arrays $\approx$ lists, but computationally much more efficient
- To compute the `y` coordinates (in an array) we need to learn about *array computations* or *vectorization*
- Array computations are useful for much more than plotting curves!

0.4 The minimal need-to-know about vectors

- Vectors are known from high school mathematics, e.g., point (x, y) in the plane, point (x, y, z) in space
- In general, a vector v is an n-tuple of numbers: $v = (v_0, \ldots, v_{n-1})$
- Vectors can be represented by lists: v_i is stored as `v[i]`, but we shall use arrays instead

0.5 Arrays can have more than one index

Just as nested lists, arrays can have multiple indices: $A_{i,j}$, $A_{i,j,k}$
Example: table of numbers, one index for the row, one for the column

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix} \qquad A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

- The no of indices in an array is the *rank* or *number of dimensions*
- Vector = one-dimensional array, or rank 1 array
- In Python code, we use Numerical Python arrays instead of nested lists to represent mathematical arrays (because this is computationally more efficient)

0.6 Storing (x,y) points on a curve in lists

Collect points on a function curve $y = f(x)$ in lists:

```
def f(x):
    return x**3

n = 5                  # no of points
dx = 1.0/(n-1)         # x spacing in [0,1]

for i in range(n):
    x.append(i*dx)
    y.append(f(x))
```

Turn lists into Numerical Python (NumPy) arrays:

```
>>> import numpy as np        # module for arrays
>>> x = np.array(xlist)      # turn list xlist into array
>>> y = np.array(ylist)
```

0.7 Make arrays directly (instead of lists)

Or drop the lists and make NumPy arrays directly:

```
>>> n = 5                      # number of points
>>> x = np.linspace(0, 1, n) # n points in [0, 1]
>>> y = np.zeros(n)           # n zeros (float data type)
>>> for i in range(n):
...     y[i] = f(x[i])
...
```

0.8 Arrays are not as flexible as list, but computationally much more efficient

- List elements can be *any* Python objects
- Array elements can only be of *one object type*
- Arrays are very efficient to store in memory and compute with if the element type is `float`, `int`, or `complex`
- Rule: use arrays for sequences of numbers!

0.9 We can work with entire arrays at once - instead of one element at a time

Compute the sine of an array:

```
from math import sin

for i in range(len(x)):
    y[i] = sin(x[i])
```

However, if `x` is array, `y` can be computed by

```
import numpy as np
y = np.sin(x)                    # x: array, y: array
```

The loop is now inside `np.sin` and implemented in very efficient C code.

Vectorization gives:

- shorter, more readable code, closer to the mathematics
- much faster code

0.10 A function `f(x)` written for a number `x` usually works for array `x` too

```
from numpy import sin, exp, linspace

def f(x):
    return x**3 + sin(x)*exp(-3*x)

x = 1.2                          # float object
y = f(x)                         # y is float

x = linspace(0, 3, 10001)        # 10000 intervals in [0,3]
y = f(x)                         # y is array
```

0.11 NOTE: `math` is for numbers and `numpy` for arrays

```
>>> import math, numpy
>>> x = numpy.linspace(0, 1, 11)
>>> math.sin(x[3])
0.2955202066613396
>>> math.sin(x)
...
TypeError: only length-1 arrays can be converted to Python scalars
>>> numpy.sin(x)
array([ 0.        ,  0.09983,  0.19866,  0.29552,  0.38941,
        0.47942,  0.56464,  0.64421,  0.71735,  0.78332,
        0.84147])
```

0.12 Very important application: vectorized code for computing points along a curve

$$f(x) = x^2 e^{-\frac{1}{2}x}\sin(x - \frac{1}{3}\pi), \quad x \in [0, 4\pi]$$

Vectorized computation of $n+1$ points along the curve.

```
import numpy as np

n = 100
x = np.linspace(0, 4*pi, n+1)
y = 2.5 + x**2*np.exp(-0.5*x)*np.sin(x-pi/3)
```

0.13 New term: vectorization

- *Scalar*: a number
- *Vector* or *array*: sequence of numbers (vector in mathematics)
- We speak about scalar computations (one number at a time) versus vectorized computations (operations on entire arrays, no Python loops)

- *Vectorized functions* can operate on arrays (vectors)
- *Vectorization* is the process of turning a non-vectorized algorithm with (Python) loops into a vectorized version without (Python) loops
- Mathematical functions in Python without `if` tests automatically work for both scalar and vector (array) arguments (i.e., no vectorization is needed by the programmer)

0.14 Small quiz:

What is output from the following code? Why?

```
import numpy as np

l = [0,0.25,0.5,0.75,1]
a = np.array(l)

print(l*2)
print(a*2)
```