L1: NumPy (Minilab)

This *minilab* provides a brief introduction to the NumPy Python library. If you have previously encountered array programming, for example in Matlab, then a large part of NumPy should be familiar; if not, the most important parts to know are introduced here. If you are interested in learning more, there is a large amount of material online (such as guides, sample code, etc.). NumPy's own documentation can be found at `https://numpy.org/doc/`. As always, the documentation is a very useful tool when programming, look there first when you have questions about NumPy.

1 Brief introduction

NumPy is a Python library specially designed for numerical calculations. Central to NumPy is the multidimensional array object, *ndarray*, which is used to store multidimensional data (vectors, matrices, etc.). NumPy provides methods to create and manipulate such arrays, but also functions to efficiently perform calculations with them. Unlike a Python list, the elements of a NumPy array are stored continuously in the computer's memory. This has several consequences, including the following:

- A NumPy array has a fixed size, i.e. a fixed number of elements. If the size is changed, a new array is created and the elements are copied, which is very expensive and should be avoided.
- All elements of a NumPy array must be of the same data type (almost always, there are exceptions).
- Mathematical calculations can be performed directly on a NumPy array. For example, the sum of two NumPy arrays of the same size can be calculated with '`+`' and the cosine of all elements of a NumPy array with the function `numpy.cos`. Because of the way a NumPy array is stored in memory, these operations can be performed significantly more efficiently than equivalent calculations with lists.
- A large number of functions for linear algebra are implemented in NumPy. For example matrix-vector multiplication and matrix inversion. Furthermore, NumPy is used in many other Python libraries, such as SciPy.

The first three tasks in this lab should be done directly in the Python prompt. All sample code assumes that NumPy has been imported as follows:

```python
import numpy as np
```

2 Vectors and matrices

1. Small vectors and arrays in NumPy are created by converting the corresponding list or tuple with the function `np.array`. For example to create the vector $v = \begin{bmatrix} 3 & -1 \end{bmatrix}$ and matrix $A = \begin{bmatrix} 2 & -1 \\ -4 & 8 \end{bmatrix}$:

```
   v = np.array([3,-1])
   A = np.array([[2,-1],[-4,8]])
```

Note that a one-dimensional NumPy array is not represented as a matrix where one of the dimensions has length 1 (unlike, for example, Matlab), so in NumPy you do not distinguish between row and column vectors.

2. You will now try a number of basic operations on the NumPy arrays `A` and `v`. Try to understand what each call does, for example by performing the calculations by hand. Do the following:
   a) Execute `A[0,1]`.
   b) Execute `A[0,:]`. Also try `A[:,1]`.
   c) Execute `A.shape` and `A.size`. Do the same with `v`.
   d) Execute `2*v`.
   e) Execute `v+v`.
   f) Execute `v*v`.
   g) Execute `A*v`.
   h) Execute `np.dot(v,v)` (scalar product of v with itself).
   i) Execute `np.dot(A,v)` (matrix-vector produkt of A and v).
   j) Execute `np.cos(v)`. Also try `np.exp(A)`.
   k) Execute `np.transpose(A)` or `A.T` (they are equivalent). Try also `np.transpose(v)`.

   The usual mathematical operations `+`, `-`, `*`, `/` and `**` (power) on NumPy arrays thus correspond to *elementwise* operations, i.e. they are performed on each element separately.

3. Most often NumPy is used to handle large vectors and matrices. Then it is inappropriate to use `np.array` to convert lists or tuples. Instead, functions designed to create NumPy arrays with specific structures are used. Try the following features:
   a) Execute `np.linspace(0,1,11)`. Also try `np.linspace(0,1,101)`. Why have we chosen 11 and 101 instead of 10 and 100?
   b) Execute `np.arange(0,1,0.1)`. Note that the final value, in this case 1, is not included in the result (like `range`).
   c) Execute `np.zeros(3)`. Also try `np.zeros((5,5))`.
   d) Execute `np.eye(4)`.
   e) Execute `np.random.rand(4)`.
   f) Execute `a=10*np.arange(10)`. Then, explore what elements in the array `a` that these commands will show
      - `a[1:]`
      - `a[:-1]`
      - `a[1:-1]`
      - `a[:len(a)//2]`
      - `a[::2]`
      - `a[::3]`
      - `a[1::2]`

- `a[0:-1:2]`
- `a[1:-1:2]`

  Hence, we have `a[start:stop:interval]` (where we can skip different arguments if we want to), and index -1 is the last element in the array. Remember that Python does “left inclusive” and “right exclusive” for intervals, e.g. `a[2:5]` means that the elements with index 2, 3, 4 are returned (and not element with index 5). Read more in the documentation.

4. Run the program `list_vs_numpy.py`. The program performs a simple vector calculation with a list or NumPy array and measures the running times. Try different sizes of the vector. Pay attention to differences in runtime and clarity of code (i.e. how easy the code is to write and understand).

3 Program yourself

5. Calculate and plot the *Gauss curve*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{1}$$

   where $\sigma = 0.1$, $\mu = 0.5$ and $x \in [0, 1]$. Define x as a NumPy array, compute $f(x)$ with NumPy functions, and plot with matplotlib. The constant π can be imported from `math` by calling `from math import pi`.