Computational Programming with Python

Unit 6: Linear Algebra

Numerical Analysis, Lund University

Lecturers: Claus Führer, Alexandros Sopasakis

2018
Vector Operations vs List Operations

As you may have noticed the operations on lists are not exactly what you would expect if you are used to vectors.

Example

```python
l1 = [1., 3.]  # defining a list
l2 = [2., 0.]  # another list
l = l1+l2      # defining a new list l after adding the two previous lists
print(l)       # prints: [1., 3., 2., 0.] which is not what you expected if you are used to linear algebra and vectors
```

Therefore, to make better sense of linear algebra constructions and operations, we discuss the datatype: array
The array datatype

Definition
An array represents a vector or a matrix in linear algebra. It is often initialised from a list or another vector. Operations +, *, /, - are all elementwise. dot is used for the scalar product.

Examples

```
vec = array([1., 3.])  # vector initialized from list
2*vec  # array([2., 6.])  # multiply a vector with 2
vec*vec  # array([1., 9.])  # multiply two vectors
vec/2  # array([0.5, 1.5])  # divide a vector by 2
vec+vec  # array([2., 6.])  # adding two vectors
```
v1 = array([1., 2., 3.]) # don’t forget the dots!
v2 = array([2, 0, 1.]) # one dot is enough

# Try the following examples below ...

v1 + v2; v1 / v2;

v1 - v2; v1 * v2

3*v1
3*v1 + 2*v2
Vectors are not lists!

Operations are not the same:

- Operations $+$ and $*$ are different
- More operations are defined: $-$, $/$
- Many functions act elementwise on vectors: $\exp$, $\sin$, $\sqrt{}$, etc.
- Vectors have a fixed size: no append method
- Only one type throughout the whole vector
  (usually float, complex, int or bool)
Similarity between Vectors and Lists

- Access vectors via their indices

```python
v = array([1., 2., 3.])
v[0] # 1.
```

- Creating Vectors - linspace The `linspace` method is a convenient way to create equally spaced arrays.

```python
xs = linspace(0, 10, 200) # 200 points between 0 and 10
xs[0] # the first point is 0
xs[-1] # the last is 10
```

So for example the plot of the sine function between 0 and 10 will be obtain by:

```python
plot(xs, sin(xs))
```
Creating Vectors (zeros, ones, …)

Some handy methods to quickly create vectors:

- **zeros** `zeros(n)` creates a vector of size `n` filled with zeros
- **ones** `ones(n)` is the same filled with ones
- **rand** `rand(n)` creates a vector with uniformly distributed random numbers between 0 and 1
- **empty** `empty(n)` creates an “empty” vector of size `n`

(try it!)
Slicing

► Parts of vectors using slices

```python
v = array([1., 2., 3.])
v[:2]  # array([1., 2.])
v[:2] = [10, 20]  # Replace a part of v
v1[:2] = [1,2,3]  # error!
v  # array([10., 20., 3.])
``` 

► Note!!! -> Vector slices are views:

```python
v = array([1., 2., 3.])
v1 = v[:2]  # v1 is now the array([1.,2.])
v1[0] = 0.  # if v1 is changed ...
v  # v is changed too! Try it yourself!
```
Vectors Functions & Vectorized Operations

- Elementwise computation: cos, sin, log, etc...

```python
cos(v1)  # works elementwise - try it!
```

- The scalar product for two vectors \( \vec{v}_1 = (1, -2) \), \( \vec{v}_2 = (3, 2) \) in linear algebra is given as \( v_1 \cdot v_2 = 1 \cdot 3 + (-2) \cdot 2 = -1 \).

```python
dot(vec, vec)  # returns 14
```

- In mathematics the Euclidean length of a vector \( \vec{v} = (a, b) \) is defined as \( |\vec{v}| = \sqrt{a^2 + b^2} \).

We compute this with the command `norm`.

```python
v = array([1., 2., 3.])
norm(vec)  # returns \( \sqrt{14} \)
len(v)  # returns 3
```

The command `len` gives us the number of elements in \( \vec{v} \).
Vectorizing Functions - later

Note that *not all functions* may be applied on vectors. For instance this one:

```python
def const(x):
    # x can be a vector of n elements
    return 1  # the result is a single element!
```

We will see later how to automatically *vectorize* a function so that it works componentwise on vectors.
Concatenating Vectors

There is no append for vectors. Instead we use: \texttt{hstack} or \texttt{vstack}

\texttt{hstack([v1, v2,..., vn])}
concatenates \textit{horizontally} the vectors \textit{v1, v2, ..., vn}.

Example:

\textbf{Symplectic permutation (a complicated math object)}
Consider a vector \(x\) of size \(2n\). We want to permute the first half with the second half of the vector and change the sign:

\[
(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_{2n}) \mapsto (-x_{n+1}, -x_{n+2}, \ldots, -x_{2n}, x_1, \ldots, x_n)
\]

\begin{verbatim}
def symp(v):
    n = len(v) // 2  # use the integer division //
    return hstack([-v[-n:], v[:n]])
\end{verbatim}
Matrices are Arrays - Created as Lists of Lists

Definition
Matrices are represented by arrays of lists of rows, which are lists as well (see example below).

```python
# the identity matrix in 2D
id = array([[1., 0.], [0., 1.]])
# Python allows this:
id = array([[1., 0.],
            [0., 1.]])
# which is more readable
```
Accessing Matrix Entries

Matrix coefficients are accessed with two indices:

\[
M = \text{array}([[1., 2.],[3.,4.]]),
\]
\[
M[0,0] \ # \text{first row, first column: 1.}
\]
\[
M[-1,0] \ # \text{last row, first column: 3.}
\]
Easily Creating Matrices

Some convenient methods to create matrices are:

- **eye** `eye(n)` is the identity matrix of size $n$
- **zeros** `zeros([n,m])` fills an $n \times m$ matrix with zeros
- **rand** `rand(n,m)` is the same with random values
- **empty** `empty([n,m])` same with “empty” values
Shape

The *shape* of a matrix is the tuple of its dimensions. The shape of an $n \times m$ matrix is $(n,m)$. It is given by the method `shape`:

```python
M = eye(3)
M.shape # (3, 3)

V = array([1., 2., 1., 4.])
V.shape # (4,) <- tuple with one element
```

Tip:

`zeros(A.shape)` returns a matrix of the same shape as A containing only zeros.

`rand(*A.shape)` does the same but with random values

Recall the difference between the arguments

`A.shape` and `*A.shape`
Transpose

The *transpose* of a matrix $A_{ij}$ is a matrix $B$ such that

$$B_{ij} = A_{ji}$$

By transposing a matrix you *switch* the two shape elements.

```
A = ...
A.shape # 3,4

B = A.T # A transpose
B.shape # 4,3
```

Note: $B$ is just a “view” of $A$. Altering $B$ changes $A$. 
Matrix multiplication with vector: $A x$

In mathematics this is computed as follows: $A x = \sum_j a_{ij} x_j$

▶ The dot operator:

In Python you can use the `dot` function: `dot(A, x)`

```python
angle = pi/3
A = array([[cos(angle), -sin(angle)],
            [sin(angle), cos(angle)]])  # Try it!
x = array([1., 0.])
Y = dot(A, x)  # the product $A \times x$
```
Matrix multiplication with vector: $A x$

In mathematics this is computed as follows: $A x = \sum_j a_{ij} x_j$

- **The dot operator:**
  In Python you can use the `dot` function: `dot(A,x)`

```python
angle = pi/3
A = array([[cos(angle), -sin(angle)],
           [sin(angle), cos(angle)]])) # Try it!
x = array([1., 0.])
Y = dot(A, x) # the product A x
```

- **The * operator:**
  $A * V$ is a legal operation which will be explained later on.

**Elementwise vs. matrix multiplication**

Note if you use multiplication operator $*$ then computations are always elementwise. It has nothing to do with the dot operation.
Dot product

**vector vector**

\[ s = \sum_i x_i y_i \]  \quad s = \text{dot}(x,y)

**matrix vector**

\[ y_i = \sum_j A_{ij} x_j \]  \quad y = \text{dot}(A,x)

**matrix matrix**

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]  \quad C = \text{dot}(A,B)

**vector matrix**

\[ y_j = \sum_i x_i A_{ij} \]  \quad y = \text{dot}(x,A)