Computational Programming with Python
Unit 5: Linear Algebra

Center for Mathematical Sciences, Lund University

Lecturer: Claus Führer, Alexandros Sopasakis
The array type

Lists are almost like vectors but the operations on list are not the linear algebra operation.

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.
Vector usage

**Example**

```python
vec = array([1., 3.]) # a vector in the plane
2*vec # array([2., 6.])
vec * vec # array([1., 9.])
vec/2 # array([0.5, 1.5])
norm(vec) # norm
dot(vec, vec) # scalar product
```
Vectors are similar to lists

- Access vectors via their indices

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

- The **length** of a vector is still obtained by the function `len`.

```python
len(v)  # 3
```
Vectors are similar to lists

- Access vectors via their indices
  
  ```python
  v = array([1., 2., 3.])
  v[0] # 1.
  ```

- The *length* of a vector is still obtained by the function `len`.
  
  ```python
  len(v) # 3
  ```

- Parts of vectors using slices
  
  ```python
  v[:2] # array([1., 2.])
  ```

- Replace parts of vectors using slices
  
  ```python
  v[:2] = [10, 20]
  v # array([10., 20., 3.])
  ```
Vectors are not lists!

Operations are not the same:

- Operations + and * are different
- More operations are defined: −, /
- Many functions act elementwise on vectors: \( \text{exp, sin, sqrt} \), etc.
- Scalar product with dot

- Vectors have a fixed size: no append method
- Only one type throughout the whole vector (usually float, complex, int or bool)

- Vector slices are views:

```python
v = array([1., 2., 3.])
v1 = v[:2]  # v is array([1., 2.])
v1[0] = 0.  # if v1 is changed ...
v  # ... v is changed too: array([0., 2., 3.])
```
More examples

```python
v1 = array([1., 2., 3.])  # don’t forget the dots!
v2 = array([2, 0, 1.])  # one dot is enough
v1 + v2; v1 / v2; v1 - v2; v1 * v2
3*v1
3*v1 + 2*v2

dot(v1, v2)  # scalar product
cos(v1)  # cosine, elementwise

# access
v1[0]  # 1.
v1[0] = 10

# slices
v1[:2]  # array([10., 2.])
v1[:2] = [0, 1]  # now v1 == array([0., 1., 3.])
v1[:2] = [1,2,3]  # error!
```
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(n)` creates a vector of size $n$ filled with zeros
- `ones(n)` is the same filled with ones
- `rand(n)` creates a vector with uniformly distributed random numbers between 0 and 1
- `empty(n)` creates an “empty” vector of size $n$ (try it!)
Concatenating Vectors

Since the + operation is redefined we need a means to *concatenate* vectors. This is where the command `hstack` comes to help. `hstack([v1, v2,\ldots, vn])` concatenates the vectors $v_1, v_2, \ldots, v_n$.

**Symplectic permutation**

We have a vector of size $2n$. We want to permute the first half with the second half of the vector with sign change:

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

```python
def symp(v):
    n = len(v) // 2  # use the integer division //
    return hstack([-v[-n:], v[:n]])
```

C. Führer, A. Sopasakis, Lund University, Sweden
Vectorized Functions

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

```python
def const(x):
    return 1
```

We will see later how to automatically *vectorize* a function so that it works componentwise on vectors.
Matrices as Lists of Lists

Definition
Matrices are represented by arrays of lists of \textit{rows}, which are lists as well.

```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 \textit{two} indices:

\begin{verbatim}
M = array([[1., 2.],[3.,4.]])
M[0,0]  # first row, first column: 1.
M[-1,0] # last row, first column: 3.
\end{verbatim}
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 \textit{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 \texttt{shape}:

\begin{verbatim}
M = eye(3)
M.shape # (3, 3)
V = array([1., 2., 1., 4.])
V.shape # (4,) <- tuple with one element
\end{verbatim}

Tip:

- \texttt{zeros(A.shape)} returns a matrix of the same shape as \( A \) containing only zeros.
- \texttt{rand(*A.shape)} does the same but with random values

Recall the difference between the arguments

- \texttt{A.shape} and \texttt{*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 vector multiplication

The mathematical concept of *reduction*:

\[ \sum_j a_{ij} x_j \]

is translated in Python in the function `dot`:

```python
angle = pi/3
M = array([[cos(angle), -sin(angle)],
           [sin(angle), cos(angle)]])
V = array([1., 0.])
Y = dot(M, V) # the product M.V
```
Matrix vector multiplication

The mathematical concept of reduction:

$$\sum_j a_{ij} x_j$$

is translated in Python in the function `dot`:

```python
angle = pi/3
M = array([[cos(angle), -sin(angle)],
           [sin(angle), cos(angle)]])
V = array([1., 0.])
Y = dot(M, V)  # the product M.V
```

Elementwise vs. matrix multiplication

The multiplication operator `*` is always elementwise. It has nothing to do with the dot operation. A*V is a legal operation which will be explained later on.
Dot product

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

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

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

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