Array computing and plotting
Part I

Foundation of programming (CK0030)

Francesco Corona

FdP

- ☻ Intro to variables, objects, modules, and text formatting
- ☻ Programming with WHILE- and FOR-loops, and lists
- ☻ Functions and IF-ELSE tests

- ☻ Data reading and writing
- ☻ Error handling
- ☹ Making modules

- ☹ **Arrays and array computing**
- ☹ Plotting curves and surfaces

FdP

A `list object` is a handy device for storing tabular data

- As in a sequence of objects or a table of objects

An array is another device which is very similar to a list

- Computationally more efficient
- Less flexible

When performing mathematical calculations with a computer, we often end up with a huge amount of numbers and their associated operations

- Storing numbers in lists may lead to slow programs
- Storing numbers in arrays can make them run faster

FdP (cont.)

This is crucial for advanced applications of mathematics in science and industry, where computer code may run for hours to weeks

Any technique that reduces execution time by some factor is important

- Focus is on clear, well-designed, and easy-to-understand code
- Then, one can start thinking about optimisation for efficiency

Arrays contribute to clear code, correctness and speed altogether

FdP (cont.)

This part of the course gives an introduction to arrays

- What they can be used for
- How they are created

FdP (cont.)

Array computing usually ends up with a lot of numbers

It may be hard to understand what these numbers mean by only looking at them

- To understand, we visualize them

We concentrate on visualising curves that reflect functions of one variable

- Curves of the form $y = f(x)$

We use arrays to store information about points on curves

Vectors
Array computing and plotting

Vectors

We start with a brief introduction to the vector concept, assuming that you know about vectors on the plane and vectors in space

This background will be valuable as we start working with arrays and plotting

Definitions, arithmetics and vector functions

Vectors

Definitions, arithmetics and vector functions

Some mathematical quantities are associated with a set of numbers

- One example is a point in the plane

To describe the point in space, we use two coordinates (real numbers)

- We name the two coordinates of a particular point as x and y
- It is common to use the notation (x, y) to denote the point
- That is, we group the numbers inside parentheses

Instead of symbols (x and y), we may use the numbers directly

- Also $(0, 0)$ and $(1.5, -2.35)$ are coordinates in the plane

Definitions, arithmetics and vector functions (cont.)

A point in a three-dimensional space has three coordinates

- We can name them x_1, x_2, and x_3

Common notation: Numbers inside parentheses, (x_1, x_2, x_3)

Alternatively, we may use the symbols x, y, and z

- We write the point as (x, y, z)
- Or, or we use numbers instead of symbols

Definitions, arithmetics and vector functions (cont.)

There problems that are formulated as n equations with n unknowns

The solution of such problems contains n numbers, that we can collect inside parentheses and number from 1 to n, as $(x_1, x_2, x_3, \ldots, x_{n-1}, x_n)$

Quantities such as (x, y), (x, y, z), or $(x_1, \ldots, , x_n)$ are called **vectors**

Definitions, arithmetics and vector functions (cont.)

Visually, a vector is an arrow that goes from the origin to a point

- Vector (x, y) is an arrow from $(0, 0)$ to the point with coordinates (x, y) in the plane
- Vector (x, y, z) is an arrow from $(0, 0, 0)$ to point (x, y, z) in three-dimensional space

Definitions, arithmetics and vector functions (cont.)

In spaces with dimension n higher than three, $(x_1, \ldots, x_n)$ a vector is a point, or an arrow in nD-space from origin $(0, \ldots, 0)$ to $(x_1, \ldots, x_n)$

We say that $(x_1, \ldots, x_n)$ is a n-vector or a vector with n components

- Each of the numbers x_1, x_2, $\ldots$ is a component or an element
- We refer to the first component (or element), the second component (or element), ...

Definitions, arithmetics and vector functions (cont.)

Python code may use a `list` or a `tuple objects` to represent a vector

```
v1 = [x, y]                                    # list of variables
v2 = (-1, 2)                                    # tuple of numbers
v3 = (x1, x2, x3)                             # tuple of variables

from math import exp
v4 = [exp(-i*0.1) for i in range(150)]
```

- `v1` and `v2` are vectors in the plane
- `v3` is a vector in a three-dimensional space
- `v4` is a vector in a 150D-space, values of the exponential function

Definitions, arithmetics and vector functions (cont.)

Remark

`List`/`tuple objects` use 0 as first index, vector (x_1, x_2) will (x_0, x_1)

- This is not common in math, but here it helps making the distance from a mathematical formulation to its Python counterpart shorter

Definitions, arithmetics and vector functions (cont.)

It is impossible to visually show what a 150D-space looks like

Going from 2D to 3D gives an idea of what it means to add dimensions

- If we forget visual perception of space, the mathematics is simple
- Going from a 4-dimensional vector to a 5-dimensional vector is as easy as adding an element to a list of symbols or numbers

Definitions, arithmetics and vector functions (cont.)

Vectors can be viewed as arrows having a length and a direction

This makes these objects very useful in geometry and physics

- The velocity of a car has a magnitude and a direction, so has the acceleration, and the position of a point in the car is also a vector
- An edge of a triangle can be viewed as a line with direction and length

Definition, arithmetics and vector functions (cont.)

Geometric/physical applications of vectors require operations on vectors

- We shall review some of the most important operations on vectors

The goal is not to teach computations with vectors, rather to illustrate how such computations are defined by math rules

Definitions, arithmetics and vector functions (cont.)

Definition

Given two vectors, (u_1, u_2) and (v_1, v_2), we can

- add the vectors $(u_1, u_2) + (v_1, v_2) = (u_1 + v_1, u_2 + v_2)$
- subtract them $(u_1, u_2) - (v_1, v_2) = (u_1 - v_1, u_2 - v_2)$

Definition

A vector (v_1, v_2) can be multiplied by a scalar a

- $a \cdot (v_1, v_2) = (av_1, av_2)$

Definitions, arithmetics and vector functions (cont.)

Definition

The scalar product (or inner product, or dot product) between two vectors (u_1, u_2) and (v_1, v_2) is a scalar[1]

- $(u_1, u_2) \cdot (v_1, v_2) = u_1 v_1 + u_2 v_2$

Definition

The length of (v_1, v_2) is $||(v_1, v_2)|| = \sqrt{(v_1, v_2) \cdot (v_1, v_2)} = \sqrt{v_1^2 + v_2^2}$

[1] It is often expressed as the product of the lengths of the two vectors multiplied by the cosine of the angle between them.

Definitions, arithmetics and vector functions (cont.)

The same mathematical operations apply to n-dimensional vectors

Remark

Instead of counting indices from 1, as we do in math,
from now on we count from 0, as we do in Python

Definition, arithmetics and vector functions (cont.)

Definition

The addition and subtraction of two vectors with n components

$$(u_0, u_1, \dots, u_{n-1}) + (v_0, v_1, \dots, v_{n-1}) = (u_0 + v_0, \dots, u_{n-1} + v_{n-1})$$
$$(u_0, u_1, \dots, u_{n-1}) - (v_0, v_1, \dots, v_{n-1}) = (u_0 - v_0, \dots, u_{n-1} - v_{n-1})$$

Definition

Multiplication of a vector $(v_0, v_1, \dots, v_{n-1})$ by a scalar a

$$a \cdot (v_0, v_1, \dots, v_{n-1}) = (av_0, av_1, \dots, av_{n-1})$$

Definitions, arithmetics and vector functions (cont.)

Definition

The inner product of two vectors $(u_0, u_1, \ldots, u_{n-1})$ and $(v_0, v_1, \ldots, v_{n-1})$

$$\begin{aligned}(u_0, u_1, \ldots, v_{n-1}) \cdot (v_0, v_1, \ldots, v_{n-1}) &= u_0 v_0 + u_1 v_1 + \cdots + u_{n-1} v_{n-1} \\ &= \sum_{j=0}^{n-1} u_j v_j\end{aligned}$$

Definition

The length $||v||$ of the n-vector $v = (v_0, v_1, \ldots, v_{n-1})$

$$\begin{aligned}\sqrt{(v_0, v_1, \ldots, v_{n-1}) \cdot (v_0, v_1, \ldots, v_{n-1})} &= (v_0 v_0 + v_1 v_1 + \cdots + v_{n-1} v_{n-1})^{\frac{1}{2}} \\ &= (v_0^2 + v_1^2 + \cdots + v_{n-1}^2)^{\frac{1}{2}} \\ &= \Big(\sum_{j=0}^{n-1} v_j^2\Big)^{\frac{1}{2}}\end{aligned}$$

Definitions, arithmetics and vector functions (cont.)

In addition to these operations, we can define other vector operations

- These are very useful for speeding up programs

Such vector operations are rarely treated in math textbooks, yet they play a significant role in some mathematical computing environments

Definition, arithmetics and vector functions (cont.)

Definition

Applying a mathematical function of one variable, say $f(x)$, to a vector is defined as a vector where f is applied to each element

Let $v = (v_0, \dots, v_{n-1})$ be a vector, $f(v) = \Big(f(v_0), \dots, f(v_{n-1})\Big)$

Example

The sine of vector v is vector $\sin(v) = \Big(\sin(v_0), \dots, \sin(v_{n-1})\Big)$

Definition, arithmetics and vector functions (cont.)

Example

It follows that squaring a vector, or more generally, raising a vector to some power, can be defined as applying the operation to each element

$$v^b = (v_0^b, \ldots, v_{n-1}^b)$$

Definitions, arithmetics and vector functions (cont.)

Definition

Another common operation between two vectors that arises in computer coding of mathematics is the 'asterisk' multiplication

$$u * v = (u_0 v_0, u_1 v_1, \cdots, u_{n-1} v_{n-1})$$

Definition

Adding a scalar a to a vector v or array is defined as adding the scalar to each component of the vector or element of the array

If a is a scalar and v a vector, $a + v = (a + v_0, \cdots, a + v_{n-1})$

Definitions, arithmetics and vector functions (cont.)

Example

How to compute a compound vector expression like $v^2 * \cos(v) * e^v + 2$

We use normal rules of math, term by term, left to right, powers before multiplications and divisions, evaluated prior to addition and subtraction

1. First, we calculate v^2, which results in a vector u
2. Then, we calculate $\cos(v)$ and call the result p
3. Then, we multiply $u * p$ to get vector w
4. The next step is to evaluate e^v, call the result q
5. This is followed by the multiplication $w * q$, whose result is r
6. Then, we add $r + 2$ to get the final result

Definition, arithmetics and vector functions (cont.)

$$\underbrace{(v)^2}_{\underbrace{u}_{1}} * \underbrace{\cos(v)}_{\underbrace{p}_{2}} * \underbrace{(e)^v}_{\underbrace{q}_{4}} + 2$$

$$\overbrace{\underbrace{w = u * p}_{3}}$$

$$\overbrace{r = \underbrace{w * q}_{5}}$$

$$\overbrace{\underbrace{r + 2}_{6}}$$

Definitions, arithmetics and vector functions (cont.)

We can, alternatively, introduce the function $f(x) = x^2 \cos(x) e^x + 2$ and use the result that $f(v)$ means applying f to each element in v

- The result is the same as in the vector expression

Arrays
Array computing and plotting

Arrays

Arrays are used to represent vectors in a program

- We introduce array programming in Python

Arrays (cont.)

First we create some lists and show how arrays differ from lists

Example

Suppose we have some function $f(x)$, we want to evaluate it at a number n of points $x_0, \cdots, x_{n-1}$

- We could collect all n pairs $(x_i, f(x_i))$ in a list, for $i = 0, \ldots, n-1$
- We could collect all x_i values and all $f(x_i)$ values in two lists, for $i = 0, \ldots, n-1$

Arrays (cont.)

An interactive session shows how to create these lists

```
>>> def f(x):
 ... return x**3                                    # sample function
 ...

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

>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]

>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here, the list elements consist of objects that are of the same type

- Any element in `pairs` is a list of two `float objects`
- Any element in `xlist` or `ylist` is a `float object`

Remark

We used list comprehensions for compactness

Arrays (cont.)

Remark

Lists are more flexible than that, because elements can be any object

```
mylist = [2, 6.0, 'tmp.pdf', [0,1]]
```

`mylist` holds an `int`, a `float`, a `string`, and a `list object`

- The combination of object types gives heterogeneous lists

We can remove/add elements from a list, anywhere in the list

Arrays (cont.)

This flexibility of lists is in general convenient to have as a programmer

- In cases where all elements are of the same type and the number of elements is fixed, arrays can be used instead

The benefits of arrays are faster computations, less memory demands

- Extensive support for mathematical operations on data

Arrays (cont.)

Because of greater efficiency/convenience, arrays are largely used

- Arrays are prominent in other programming languages

Lists are the choice when we need the flexibility of adding/removing elements or when the elements may be of different object types

Numerical Python arrays

Arrays

Numerical Python arrays (cont.)

An `array object` can be viewed as a variant of a list

- All elements must be of the same type (preferably integer, real, or complex numbers), for efficient numerical computing and storage
- The number of elements must be known when the array is created [2]
- Arrays with one index are often called vectors
- Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists
- Arrays can also have three or more indices

[2] The number of elements in an array can be changed, at a large computational cost

Numerical Python arrays (cont.)

Arrays are not in Python, package Numerical Python (`numpy`) is needed

- With `numpy`, a wide range of math operations can be done directly on arrays (no need for loops over array elements)
- This process is commonly called **vectorization**

Numerical Python arrays (cont.)

Remark

There is an actual object type called `array` in standard Python

- This data type is not so efficient for math computations

Numerical Python arrays (cont.)

The standard import statement for Numerical Python

```
import numpy as np
```

Example

To convert a list r to an array a, we use function array from numpy

```
a = np.array(r)
```

Numerical Python arrays (cont.)

Example

To create a new array of length `n` which is filled with zeros

```
a = np.zeros(n)
```

By default, the array elements are `float objects`

- To specify elements of a different type (e.g., zeros that are `int objects`), a second argument can be given

Numerical Python arrays (cont.)

Example

Function `zeros_like` generates an array of zeros where the length is that of the array `c` and the element type is the same as those in `c`

```
a = np.zeros_like(c)
```

Example

An array of `n` elements equally spaced in an interval `[p,q]`

- The numpy function `linspace` creates such an array

```
a = np.linspace(p, q, n)
```

Numerical Python arrays (cont.)

Array elements are accessed by square brackets as for lists: `a[i]`

Slices also work as for lists

- `a[1:-1]` picks all elements of `a` except the first and the last
- Contrary to lists, `a[1:-1]` is NOT a copy of the data in `a`

Example

```
a = np.zeros(n)

b = a[1:-1]
b[2] = 0.1
```

The second instruction will change `b[2]` and also `a[3]` to `0.1`

Numerical Python arrays (cont.)

A slice `a[i:j:s]` picks the elements starting with index `i`, stepping `s` indices at the time up to, but not including, `j`

- Omitting `i` implies `i=0`
- Omitting `j` implies `j=n`, if `n` is the number of elements in the array

Example

`a[0:-1:2]` picks every two elements up to, but not including, the last element, while `a[::4]` picks every four elements in the whole array

Numerical Python arrays (cont.)

Remark

```
import numpy as np
```

This statement with subsequent prefixing of NumPy functions/variables by np, is standard syntax in the Python scientific computing community

```
from numpy import *
```

To make Python programs look closer to MATLAB and ease transition to/from that language, this statement is used to remove the prefix

- This is the present standard in interactive Python shells

Example

For $f(x) = \sinh(x-1) * \sin(wt)$

```
from numpy import sinh, sin

def f(x):
 return sinh(x-1)*sin(w*t)
```

Function values

Arrays

Function values

With basic operations, we can make arrays from lists `xlist` and `ylist`

Example

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

>>> n = 5
>>> dx = 1.0/(n-1)

>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]


>>> import numpy as np
>>> x2 = np.array(xlist)        # turn list xlist into array x2
>>> y2 = np.array(ylist)        # turn list ylist into array y2

>>> x2
    array([0., 0.25, 0.5, 0.75, 1.])

>>> y2
    array([0., 0.015625, 0.125, 0.421875, 1.])
```

Function values

Making a list and then converting the list to an array is not optimal

We can compute the arrays directly

- The equally spaced coordinates in `x2` can be computed by the `np.linspace` `function`
- The `y2` array can be first created by `np.zeros`, to ensure that `y2` has the right length `len(x2)`
- Then, we can run a `FOR-loop` to fill in all elements in `y2` with `f` values

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

>>> n = len(xlist)
>>> x2 = np.linspace(0, 1, n)

>>> y2 = np.zeros(n)
>>> for i in xrange(n):                    # xrange vs range
 ... y2[i] = f(x2[i])

>>> y2
    array([ 0. , 0.015625, 0.125 , 0.421875, 1. ])
```

Function values (cont.)

In the `FOR-loop`: `xrange` instead of `range`

Remark

The former (`xrange`) is faster for long loops because it avoids generating and storing a list of integers, as it generates the values one by one

- For loops over long arrays, `xrange` is preferable over `range`

In Python version 3.x, `range` is the same as `xrange`

Creating an array of a given length is referred to as **allocating** the array

- Part of the computer's memory (RAM)
  is marked for being occupied by the array
- Arrays can fill up most of the memory

Function values (cont.)

Example

To shorten the code, create the `y2` data in a `list comprehension`

```
>>> x2 = np.linspace(0, 1, n)

>>> y2 = np.array([f(xi) for xi in x2])
```

- `List comprehensions` produce lists, not arrays
- Transform `list objects` to `array objects`

Vectorisation

Arrays

Vectorisation

A great advantage with arrays is that we can eliminate loops

- Loops over very long arrays may run slowly
- We can apply `f` directly to the whole array

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

>>> x2
    array([ 0.  , 0.25, 0.5 , 0.75, 1. ])

>>> y2 = f(x2)

>>> y2
    array([ 0.  , 0.015625, 0.125  , 0.421875, 1. ])
```

Vectorisation (cont.)

Remark

Instead of calling `f(x2)`, we can write the formula `x2**3` directly

The important point is that `numpy` implements vector arithmetics

- For arrays of arbitrary dimensions

Also, `numpy` provides its own versions of mathematical functions

- `cos`, `sin`, `exp`, `log`, ...

So, with array arguments, the function is applied to each element

Vectorisation (cont.)

This code computes each array element separately

Example

```
from math import sin, cos, exp
import numpy as np

x = np.linspace(0, 2, 201)

r = np.zeros(len(x))
for i in xrange(len(x)):
 r[i] = sin(np.pi*x[i])*cos(x[i])*exp(-x[i]**2)+2+x[i]**2
```

Vectorisation (cont.)

This version of the code operates on arrays directly

Example

```
x = np.linspace(0, 2, 201)

r = np.sin(np.pi*x)*np.cos(x)*np.exp(-x**2)+2+x**2
```

Equivalently, without the np prefix

```
from numpy import sin, cos, exp, pi

r = sin(pi*x)*cos(x)*exp(-x**2)+2+x**2
```

Vectorisation (cont.)

Remark

It is important to understand that the `sin` `function` from `math` `module` is different from the `sin` `function` provided by `numpy`

- The former does not allow array arguments
- The latter accepts real numbers and arrays

Vectorisation (cont.)

Replacing a loop for computing `r[i]` with a vector/array expression as `sin(x)*cos(x)*exp(-x**2) + 2 + x**2` is called **vectorization**

```
r = sin(pi*x)*cos(x)*exp(-x**2) + 2 + x**2
```

The loop version can be referred to as **scalar code**

```
x = np.linspace(0, 2, 201)

r = np.zeros(len(x))
for i in xrange(len(x)):
 r[i] = sin(np.pi*x[i])*cos(x[i])*exp(-x[i]**2)+2+x[i]**2
```

Vectorisation (cont.)

Example

Scalar version

```
import numpy as np
import math

x = np.zeros(N); y = np.zeros(N)
dx = 2.0/(N-1)                          # spacing of x coordinates

for i in range(N):
 x[i] = -1 + dx*i
 y[i] = math.exp(-x[i])*x[i]
```

Vector version

```
x = np.linspace(-1, 1, N)
y = np.exp(-x)*x
```

Vectorisation (cont.)

`List comprehension` result in scalar code, as there are explicit `FOR-loops` operating on scalar quantities

```
x = array([-1 + dx*i for i in range(N)])
y = array([np.exp(-xi)*xi for xi in x])
```

The requirement of vectorised code is that there are NO explicit Python `FOR-loops`

Remark

In the `numpy` package, the loops to compute array elements are performed in fast C/Fortran code :)

Vectorisation (cont.)

`Functions` for scalar arguments automatically work for array arguments provided the inner `functions` in the definition accept array arguments

Example

```
def f(x):
 return x**4*exp(-x)

x = np.linspace(-3, 3, 101)
y = f(x)
```

`Function` `exp` must be imported as `from numpy import *`

- Or explicitly as `from numpy import exp`

One can prefix `exp` as in `np.exp` (less attractive math syntax)

Vectorisation (cont.)

When `function` `f(x)` works with array arguments `x`, `f` is vectorised

Provided the math expressions in `f` involve arithmetic operations and basic math functions from `math module`, `f` will be automatically vectorised by importing the `functions` from `numpy` instead of `math`

- Note true if the expressions inside `f` involve `IF-tests`
- To work with arrays, that code need to be re-written

Remark

Vectorisation helps speeding up code performing computations on arrays

Moreover, vectorisation gives more compact code, which is easier to read

Numpy arrays

Array computing and plotting

Numpy arrays

Some more advanced but useful operations with `numpy` arrays

Copying arrays
Numpy arrays

Copying arrays (cont.)

Example

Let x be an array, changing a will also affect x

```
>>> import numpy as np

>>> x = np.array([1, 2, 3.5])

>>> a = x
>>> a[-1] = 3                        # this changes x[-1] too!

>>> x
    array([ 1., 2., 3.])
```

Statement a = x makes a refer to the same array as x

Copying arrays (cont.)

Example

Changing `a` without changing `x` requires `a` to be a copy of `x`

```
>>> a = x.copy()
>>> a[-1] = 9

>>> a
    array([ 1., 2., 9.])

>>> x
    array([ 1., 2., 3.])
```

In-place arithmetics

Numpy arrays

In-place arithmetics

Let `a` and `b` be two arrays of the same shape

- The expression `a += b` means `a = a + b`

In the statement `a = a + b`, the sum `a + b` is first computed, this yields a new array, and then the name `a` is bound to this new array

- Old array `a` is lost, unless there are other names assigned to it

In-place arithmetics

In `a += b`, elements of `b` are added directly to the elements of `a`

- There is no hidden intermediate array as in `a = a + b`

This implies that `a += b` is more efficient than `a = a + b`

- As Python avoids making an extra array

Operators `+=`, `*=`, etc perform **in-place arithmetics** on arrays

In-place arithmetics (cont.)

On the computation of a compound array expression

Example

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

1. `r1 = x**4`
2. `r2 = 3*r1`
3. `r3 = 2*x`
4. `r4 = r2 + r3`
5. `r5 = r4 + 4`
6. `r6 = x + 1`
7. `r7 = r5/r6`
8. `a = r7`

In-place arithmetics (cont.)

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

- A significantly less readable code

```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

In-place arithmetics (cont.)

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

–

- `r1 = x**4`
- `r2 = 3*r1`
- `r3 = 2*x`
- `r4 = r2 + r3`
- `r5 = r4 + 4`
- `r6 = x + 1`
- `r7 = r5/r6`
- `a = r7`

- `a = x.copy()`
- `a **= 4`
- `a *= 3`

  –
- `a += 2*x`
- `a += 4`

  –
- `a /= x + 1`

  –

With in-place arithmetics we can save creating three new arrays

- The three arrays are from copying `a`
  and computing the RHS `2*x` and `x+1`

In-place arithmetics (cont.)

Remark

Often in computational sciences, a large number of arithmetics is performed on big arrays, saving memory/array allocation time

- In-place arithmetics becomes important

In-place arithmetics (cont.)

Mixing up assignment and in-place arithmetics makes it easier (worse) to make unintended changes to more than one array

Example

This code changes `x`, as `a` refers to the same array as `x`

```
a = x
a += y
```

- The change of `a` is done in-place

Array allocation
Numpy arrays

Array allocation

Function `np.zeros` is handy for making a new array `a` of given size

- Often size and type of array elements have to match some existing array `x`

We can either copy the original array

```
a = x.copy()
```

or, we can fill in new elements in `a`

```
a = np.zeros(x.shape, x.dtype)
```

Attribute `x.dtype` holds the array element type (`dtype` for data type), `x.shape` is a tuple with the array dimensions

- Variable `a.ndim` holds the number of dimensions

Array allocation (cont.)

The `np.asarray` function turns an object into an array

```
a = np.asarray(a)
```

Nothing is copied if `a` is already is an array

If `a` is a list or tuple, a new array with a copy of the data is created

Generalised indexing

Numpy arrays

Generalised indexing

Slices can be used to extract and manipulate subarrays

The slice `f:t:i` corresponds to the index set

- `f`, `f+i`, `f+2*i`,`...` up to, but not including, `t`

Such an index set can be given explicitly too

- `a[range(f,t,i)]`

The integer list from `range` can be used as a set of indices

Generalised indexing (cont.)

Any integer list or integer array can be used as index

```
>>> a = np.linspace(1, 8, 8)
>>> a
    array([ 1., 2., 3., 4., 5., 6., 7., 8.])

>>> a[[1,6,7]] = 10
>>> a
    array([ 1., 10., 3., 4., 5., 6., 10., 10.])


>>> a[range(2,8,3)] = -2                # same as a[2:8:3] = -2
>>> a
    array([ 1., 10., -2., 4., 5., -2., 10., 10.])
```

Generalised indexing (cont.)

We can also use boolean arrays to generate an index set

- The indices in the set correspond to indices for which the boolean array has `True` values

Example

```
>>> a[a < 0]                    # pick out the negative elements of a
    array([-2., -2.])


>>> a[a < 0] = a.max()
>>> a
    array([ 1., 10., 10., 4., 5., 10., 10., 10.])


>>>                # Replace elements where a is 10 by the first
>>>                              # elements from another array/list
>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]
>>> a
    array([ 1., 10., 20., 4., 5., 30., 40., 50.])
```

Generalised indexing (cont.)

Remark

This functionality allows expressions, like `a[x<m]`

Array type, generation and manipulation

Numpy arrays

Array type, generation and manipulation

To check an object's type, it is possible to use the `type` function

- In case of a `numpy` array, the type is a `ndarray object`

Example

```
>>> a = np.linspace(-1, 1, 3)
>>> a
    array([-1., 0., 1.])

>>> type(a)
    <type 'numpy.ndarray'>
```

Array type, generation and manipulation (cont.)

To check if a variable is a `ndarray` or a `float` or a `int object`

- It is possible to use `function isinstance`

Example

```
>>> isinstance(a, np.ndarray)
    True

>>> isinstance(a, (float,int))                    # float or int?
    False
```

Array type and generation (cont.)

Example

Suppose we have some constant function and we want to vectorise it

```
def f(x):
 return 2
```

The function accepts an array argument x and returns a float object

- The vectorised version should return an array of the same shape of x, with each element equal 2

```
def fv(x):
 return np.zeros(x.shape, x.dtype) + 2
```

Yet, an optimal function works with both scalar and array arguments

Array type and generation (cont.)

To handle this possibility, we must test for the argument type

```
def f(x):
 if isinstance(x, (float, int)):
  return 2

 elif isinstance(x, np.ndarray):
  return np.zeros(x.shape, x.dtype) + 2

 else:
  raise TypeError\
  ('x must be int, float or ndarray, not %s' % type(x))
```

Array type and generation (cont.)

A special compact syntax `r_[f:t:s]` for the `linspace` function

Example

```
>>> a = r_[-5:5:11j]            # same as linspace(-5, 5, 11)
>>> print a
    [-5.  -4.  -3.  -2.  -1.  0.  1.  2.  3.  4.  5.]
```

- `11j`: 11 elements (between −5 and 5, including upper limit 5)
- That is, the number of elements in the array is given with the imaginary number syntax

Shape manipulation

Numpy arrays

Shape manipulation

The shape attribute in array objects holds the shape

- The size of each dimension of an array

Function size returns the total number of elements

Example

```
>>> a = np.linspace(-1, 1, 6)
>>> print a
    [-1. -0.6 -0.2 0.2 0.6 1. ]

>>> a.shape
    (6,)

>>> a.size
    6
```

Shape manipulation (cont.)

A few equivalent ways of changing the shape of an array

```
>>> print a
    [-1.  -0.6 -0.2  0.2  0.6  1. ]

>>> a.shape = (2, 3)
>>> a = a.reshape(2, 3)                          # alternative
>>> a.shape
    (2, 3)

>>> print a
    [[-1.   -0.6  -0.2]
    [  0.2  0.6  1. ]]

>>> a.size                              # total no of elements
    6

>>> len(a)                                       # no of rows
    2

>>> a.shape = (a.size,)                        # reset shape
```