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: $v = (v_0, \ldots, v_{n-1})$
- There are rules for various mathematical operations on vectors, read the book for details (later?)
- Vectors can be represented by lists: v_i is stored as `v[i]`

Vectors and arrays are concepts in this chapter so we need to briefly explain what these concepts are. It takes separate math courses to understand what vectors and arrays really are, but in this course we only need a small subset of the complete story. A learning strategy may be to just start using vectors/arrays in programs and later, if necessary, go back to the more mathematical details in the first part of Ch. 4.

The minimal need-to-know about arrays

- Arrays are a generalization of vectors where we can have multiple indices: $A_{i,j}$, $A_{i,j,k}$ – in code this is nothing but nested lists, accessed as `A[i][j]`, `A[i][j][k]`
- Example: table of numbers, one index for the row, one for the column

$$\left[\begin{array}{cccc} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{array}\right] \qquad A = \left[\begin{array}{ccc} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{array}\right]$$

- 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 lists to represent mathematical arrays (because this is computationally more efficient)

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

- Collect (x, y) points on a function curve $y = f(x)$ in a list:

```
>>> def f(x):
...     return x**3         # sample function
...
>>> n = 5                   # no of points in [0,1]
>>> dx = 1.0/(n-1)          # x spacing
>>> 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)]
```

- Turn lists into Numerical Python (NumPy) arrays:

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

Make arrays directly (instead of lists)

- Instead of first making lists with x and $y = f(x)$ data, and then turning lists into arrays, we can make NumPy arrays directly:

```
>>> n = 5                      # number of points
>>> x2 = np.linspace(0, 1, n) # n points in [0, 1]
>>> y2 = np.zeros(n)           # n zeros (float data type)
>>> for i in xrange(n):
...     y2[i] = f(x2[i])
...
```

- `xrange` is similar to `range` but faster (esp. for large `n` – `xrange` does not explicitly build a list of integers, `xrange` just lets you loop over the values)
- List comprehensions create lists, not arrays, but we can do

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

The clue about NumPy arrays (part 1)

- Lists can hold any sequence of any Python objects
- Arrays can only hold objects of the same type
- Arrays are most efficient when the elements are of basic number types (`float`, `int`, `complex`)
- In that case, arrays are stored efficiently in the computer memory and we can compute very efficiently with the array elements

The clue about NumPy arrays (part 2)

- Mathematical operations on whole arrays can be done without loops in Python
- For example,

```
x = np.linspace(0, 2, 10001)   # numpy array
for i in xrange(len(x)):
    y[i] = sin(x[i])
```

  can be coded as

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

  and the loop over all elements is now performed in a very efficient C function
- Operations on whole arrays, instead of using Python for loops, is called *vectorization* and is very convenient and very efficient (and an important programming technique to master)

Vectorizing the computation of points on a function curve

- Consider the loop with computing x coordinates (`x2`) and $y = f(x)$ coordinates (`y2`) along a function curve:

```
x2 = np.linspace(0, 1, n)   # n points in [0, 1]
y2 = np.zeros(n)            # n zeros (float data type)
for i in xrange(n):
    y2[i] = f(x2[i])
```

- This computation can be replaced by

```
x2 = np.linspace(0, 1, n)   # n points in [0, 1]
y2 = f(x2)                  # y2[i] = f(x[i]) for all i
```

- Advantage: 1) no need to allocate space for `y2` (via `np.zeros`), 2) no need for a loop, 3) *much* faster computation
- Next slide explains what happens in `f(x2)`

How a vectorized function works

- Consider

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

- `f(x)` is intended for a number `x`, called *scalar* – contrary to vector/array
- What happens with a call `f(x2)` when `x2` is an array?
- The function then evaluates `x**3` for an array `x`
- Numerical Python supports arithmetic operations on arrays, which correspond to the equivalent operations on each element

```
from numpy import cos, exp
x**3                # x[i]**3                    for all i
cos(x)              # cos(x[i])                  for all i
x**3 + x*cos(x)     # x[i]**3 + x[i]*cos(x[i])   for all i
x/3*exp(-x*a)       # x[i]/3*exp(-x[i]*a)        for all i
```

Vectorization

- Functions that can operate on vectors (or arrays in general) are called vectorized functions (containing vectorized expressions)
- Vectorization is the process of turning a non-vectorized expression/algorithm into a vectorized expression/algorithm
- Mathematical functions in Python without `if` tests automatically work for both scalar and array (vector) arguments (i.e., no vectorization is needed by the programmer)

More explanation of a vectorized expression

- Consider `y = x**3 + x*cos(x)` with array `x`
- This is how the expression is computed:

```
r1 = x**3      # call C function for x[i]**3 loop
r2 = cos(x)    # call C function for cos(x[i]) loop
r3 = x*r2      # call C function for x[i]*r2[i] loop
y = r1 + r3    # call C function for r1[i]+r3[i] loop
```

- The C functions are highly optimized and run very much faster than Python `for` loops (factor 10-500)
- Note: `cos(x)` calls `numpy`'s `cos` (for arrays), not `math`'s `cos` (for scalars) if we have done `from numpy import cos` or `from numpy import *`

Summarizing array example

- Make two arrays `x` and `y` with 51 coordinates x_i and $y_i = f(x_i)$ on the curve $y = f(x)$, for $x \in [0, 5]$ and $f(x) = e^{-x}\sin(\omega x)$:

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

def f(x):
    return exp(-x)*sin(omega*x)

omega = 2*pi
x = linspace(0, 5, 51)
y = f(x)    #  or  y = exp(-x)*sin(omega*x)
```

- Without `numpy`:

```
from math import exp, sin, pi

def f(x):
    return exp(-x)*sin(omega*x)

omega = 2*pi
n = 51
dx = (5-0)/float(n)
x = [i*dx for i in range(n)]
y = [f(xi) for xi in x]
```

Assignment of an array does not copy the elements!

- Consider this code:

```
a = x
a[-1] = q
```

- Is `x[-1]` also changed to `q`? Yes!
- `a` refers to the same array as `x`
- To avoid changing `x`, `a` must be a copy of `x`:

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

- The same yields slices:

```
a = x[r:]
a[-1] = q   # changes x[-1]!
a = x[r:].copy()
a[-1] = q   # does not change x[-1]
```

In-place array arithmetics

- We have said that the two following statements are equivalent:
```
a = a + b     # a and b are arrays
a += b
```
- Mathematically, this is true, but not computationally
- `a = a + b` first computes `a + b` and stores the result in an intermediate (hidden) array (say) `r1` and then the name `a` is bound to `r1` – the old array `a` is lost
- `a += b` adds elements of `b` *in-place* in `a`, i.e., directly into the elements of `a` without making an extra `a+b` array
- `a = a + b` is therefore less efficient than `a += b`

Compound array expressions

- Consider
```
a = (3*x**4 + 2*x + 4)/(x + 1)
```
- Here are the actual computations:
```
r1 = x**4;  r2 = 3*r1;  r3 = 2*x;  r4 = r1 + r3
r5 = r4 + 4;  r6 = x + 1; r7 = r5/r6; a = r7
```
- With in-place arithmetics we can save four extra arrays, at a cost of much less readable code:
```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

More on useful array operations

- Make a new array with same size as another array:

```
# x is numpy array
a = x.copy()
# or
a = zeros(x.shape, x.dtype)
```

- Make sure a list or array is an array:

```
a = asarray(a)
b = asarray(somearray, dtype=float)
```

- Test if an object is an array:

```
>>> type(a)
<type 'numpy.ndarray'>
>>> isinstance(a, ndarray)
True
```

- Generate range of numbers with given spacing:

```
>>> arange(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5])   # 1 is not included!
>>> linspace(-1, 0.5, 4)          # equiv. array

>>> from scitools.std import *
>>> seq(-1, 1, 0.5)               # 1 is included
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

Example: vectorizing a constant function

- Constant function:

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

- Vectorized version must return array of 2's:

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

- New version valid both for scalar and array `x`:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    elif isinstance(x, ndarray):
        return zeros(x.shape, x.dtype) + 2
    else:
        raise TypeError\
        ('x must be int/float/ndarray, not %s' % type(x))
```

Generalized array indexing

- Recall slicing: `a[f:t:i]`, where the slice `f:t:i` implies a set of indices
- Any integer list or array can be used to indicate a set of indices:

```
>>> a = 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.])
```

- Boolean expressions can also be used (!)

```
>>> a[a < 0]            # pick out all negative elements
array([-2., -2.])
>>> a[a < 0] = a.max() # if a[i]<10, set a[i]=10
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
```

Summary of vectors and arrays

- Vector/array computing: apply a mathematical expression to every element in the vector/array
- Ex: `sin(x**4)*exp(-x**2)`, `x` can be array or scalar, for array the `i`'th element becomes `sin(x[i]**4)*exp(-x[i]**2)`
- Vectorization: make scalar mathematical computation valid for vectors/arrays
- Pure mathematical expressions require no extra vectorization
- Mathematical formulas involving `if` tests require manual work for vectorization:

```
scalar_result = expression1 if condition else expression2
vector_result = where(condition, expression1, expression2)
```

Array functionality

`array(ld)`	copy list data `ld` to a `numpy` array
`asarray(d)`	make array of data `d` (copy if necessary)
`zeros(n)`	make a vector/array of length `n`, with zeros (`float`)
`zeros(n, int)`	make a vector/array of length `n`, with `int` zeros
`zeros((m,n), float)`	make a two-dimensional with shape (`m`,`n`)
`zeros(x.shape, x.dtype)`	make array with shape and element type as `x`
`linspace(a,b,m)`	uniform sequence of `m` numbers between `a` and `b`
`seq(a,b,h)`	uniform sequence of numbers from `a` to `b` with step `h`
`iseq(a,b,h)`	uniform sequence of integers from `a` to `b` with step `h`
`a.shape`	tuple containing `a`'s shape
`a.size`	total no of elements in `a`
`len(a)`	length of a one-dim. array `a` (same as `a.shape[0]`)
`a.reshape(3,2)`	return `a` reshaped as 2×3 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices `1`,...,`k-1`
`a[1:8:3]`	slice: reference data with indices `1`, `4`,...,`7`
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	`numpy` functions applicable to arrays
`c = concatenate(a, b)`	`c` contains `a` with `b` appended
`c = where(cond, a1, a2)`	`c[i] = a1[i]` if `cond[i]`, else `c[i] = a2[i]`
`isinstance(a, ndarray)`	is `True` if `a` is an array