5mm.

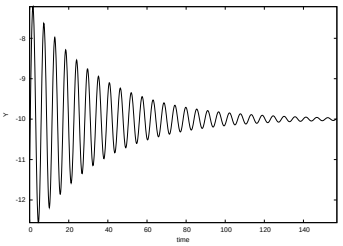
INF1100 Lectures, Chapter 5: Array Computing and Curve Plotting

Hans Petter Langtangen

Simula Research Laboratory
University of Oslo, Dept. of Informatics

Goals of this chapter (part 1)

- Learn to plot (visualize) function curves
- Learn to store points on curves in arrays (≈ lists)



Goals of this chapter (part 2)

- Curves $y = f(x)$ are visualized by drawing straight line between consequtive points along the curve
- We need to store the coordinates of the points along the curve in lists or arrays `x` and `y`
- Arrays ≈ lists, but much computationally more efficient
- To compute the `y` coordinates (in an array) we need to learn about *array computations* or *vectorization*
- Array computations are useful for much more than plotting curves!
- When we need to compute with large amounts of numbers, we store the numbers in arrays and compute with arrays – this gives shorter and faster code

The minimal need-to-know about vectors

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.

- 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]`

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

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix} \qquad A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

- 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:

```
>>> from numpy import *    # get access to key numpy functionality
>>> x2 = array(xlist)      # turn list xlist into array
>>> y2 = 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 = linspace(0, 1, n)   # n points between 0 and 1
>>> y2 = 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 = 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 = linspace(0, 2, 10001)   # numpy array
for i in xrange(len(x)):
    y[i] = sin(x[i])
```

  can be coded as

```
y = sin(x)
```

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

- This computation can be replaced by

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

- Advantage: 1) no need to allocate space for `y2` (via `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

```
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)

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)
```

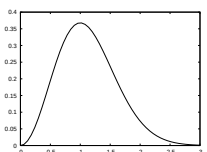
- Without `numpy`:

```
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]
```


Plotting curves; the very basics

- Having points along a curve $y = f(x)$ stored in one-dimensional arrays `x` and `y`, we can easily plot the curve by `plot(x, y)`
- Complete program:

```
from scitools.std import *  # import numpy and plotting
t = linspace(0, 3, 51)      # 51 points between 0 and 3
y = t**2*exp(-t**2)         # vectorized expression
plot(t, y)
hardcopy('tmp1.eps')  # make PostScript image for reports
hardcopy('tmp1.png')  # make PNG image for web pages
```



Plotting curves; decorating the plot

- A plot should have labels on the axis, a title, maybe specified extent of the axis:

```
from scitools.std import *  # import numpy and plotting

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = f(t)
plot(t, y)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6])   # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')
```

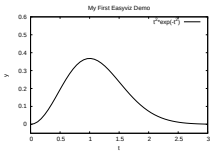
SciTools and Easyviz

- SciTools (`scitools`) is a Python package with lots of useful tools for mathematical computations, developed here in Oslo (Langtangen, Ring, Wilbers, Bredesen, ...)
- Easyviz is a subpackage of SciTools (`scitools.easyviz`) doing plotting with Matlab-like syntax
- Everything from Easyviz and NumPy gets imported by
  ```
  from scitools.std import *
  ```
- Easyviz is only a unified interface to many different plotting programs (Gnuplot, Matlab, Grace, Matplotlib, Vtk, OpenDX)
- In this course we recommend to use Gnuplot to produce the plots (because Gnuplot installs easily everywhere)
- You need to install NumPy, SciTools and Gnuplot!

Plotting curves; more compact syntax

- Instead of calling several functions for setting axes labels, legends, title, axes extent, etc., this information can be given as keyword arguments to `plot`

```
plot(t, y,
     xlabel='t',
     ylabel='y',
     legend='t^2*exp(-t^2)',
     axis=[0, 3, -0.05, 0.6],
     title='My First Easyviz Demo',
     hardcopy='tmp1.eps',
     show=True)   # display on the screen (default)
```



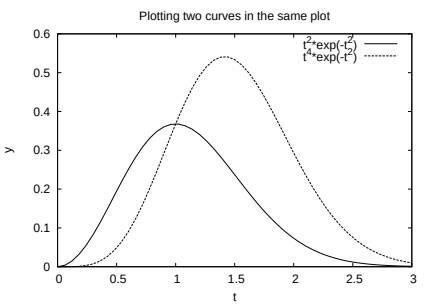
Plotting several curves in one plot

```
from scitools.std import *   # for curve plotting

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

# Matlab-style syntax:
plot(t, y1)
hold('on')   # continue plotting in the same plot
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')
```


Alternative, more compact plot command

```
plot(t, y1, t, y2,
     xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')

# equivalent to
plot(t, y1)
hold('on')
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')
```


The resulting plot with two curves



Controlling line styles

- When plotting multiple curves in the same plot, the different lines (normally) look different
- We can control the line type and color, if desired

```
plot(t, y1, 'r-')   # red (r) line (-)
hold('on')
plot(t, y2, 'bo')   # blue (b) circles (o)

# or
plot(t, y1, 'r-', t, y2, 'bo')
```

- See the book or

```
Unix/DOS> pydoc scitools.easyviz
```

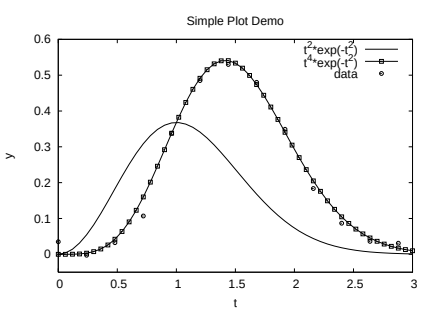
for documentation of colors and line styles

Example: two curves + random points (noise), part 1

- We want to plot the `f1` and `f2` functions, plus some noisy data points around the `f2` curve
- The noisy data points should be randomly displaced circles at every four points

```
# y1 and y2 as previous example
plot(t, y1, 'r-'); hold('on'); plot(t, y2, 'ks-')

# pick out each 4 points and add random noise:
t3 = t[::4]      # slice, stride 4
random.seed(11)  # fix random sequence
noise = random.normal(loc=0, scale=0.02, size=len(t3))
y3 = y2[::4] + noise
plot(t3, y3, 'bo')

legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t')
ylabel('y')
show()              # display screen plot (default)
hardcopy('tmp3.eps')
hardcopy('tmp3.png')
```


Example: two curves + random points (noise), part 2



Quick plotting with minimal typing

```
t = linspace(0, 3, 51)
plot(t, t**2*exp(-t**2), t, t**4*exp(-t**2))
```


Plot function given on the command line

- Task: give the function to be plotted on the command line
  ```
  Unix/DOS> python plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
  ```
- Syntax: `plotf.py expression xmin xmax`
- Program:
  ```
  from scitools.std import *
  formula = sys.argv[1]
  xmin = eval(sys.argv[2])
  xmax = eval(sys.argv[3])

  x = linspace(xmin, xmax, 101)
  y = eval(formula)
  plot(x, y, title=formula)
  ```
- Make this program more foolproof by checking input data!

Making animations (movies)

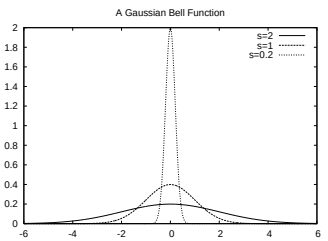
- Consider the Gaussian bell function (see next slide for a plot):

$$f(x;m,s) = \frac{1}{\sqrt{2\pi}}\frac{1}{s}\exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right]$$

- Goal: make a movie showing how $f(x)$ varies in shape as s decreases
- Idea: put many plots (for different s values) together - just as a movie made from cartoons
- Program: loop over s values, call `plot` for each s and make hardcopy, combine all hardcopies to a movie
- Very important: fix the y axis! Otherwise, it always adapts to the peak of the function and the visual impression gets wrong

Plot of the Gaussian bell function

- $f(x;m,s)$: m is the location of the peak, s is a measure of the width of the function



The code for making the animation

```
from scitools.std import *
import time

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0;  s_start = 2;  s_stop = 0.2
s_values = linspace(s_start, s_stop, 30)
x = linspace(m -3*s_start, m + 3*s_start, 1000)
# f is max for x=m; smaller s gives larger max value
max_f = f(m, m, s_stop)

# show the movie on the screen
# and make hardcopies of frames simultaneously:
frame_counter = 0
for s in s_values:
    y = f(x, m, s)
    plot(x, y, axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend='s=%4.2f' % s,
         hardcopy='tmp_%04d.eps' % frame_counter)
    frame_counter += 1
    #time.sleep(0.2)  # pause to control movie speed

movie('tmp_*.eps')    # make movie file movie.gif
```


How to play movie files

- Play animated GIF file:
  ```
  Unix/DOS> animate movie.gif
  ```
  (`animate` is a program in the ImageMagick suite)
- `movie` can also make MPEG and AVI movie formats
- To play MPEG, AVI, DV, DVD, WMV, WMA, MP4, MP3, OGG, WAV, FLAC, ... use the cross-platform VLC player

Curves in pure text (part 1)

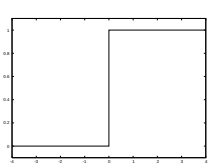
- Plots are persisently stored in image files (PostScript or PNG)
- Sometimes you want a plot in your program (e.g., to prove that the curve looks right in a compulsory exercise where only the program, not a nicely typeset report, is submitted)
- `scitools.aplotter` can then be used for drawing primitive curves in pure text (ASCII) format:
  ```
  >>> from scitools.aplotter import plot
  >>> from numpy import linspace, exp, cos, pi
  >>> x = linspace(-2, 2, 81)
  >>> y = exp(-0.5*x**2)*cos(pi*x)
  >>> plot(x, y)
  ```
- Try it out interactively!

Plotting the Heaviside function (part 1)

- Aim: show that plotting is not always straightforward
- Example: the Heaviside function is frequently used in science and engineering,

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

- Python implementation:
  ```
  def H(x):
      return (0 if x < 0 else 1)
  ```



Plotting the Heaviside function (part 2)

- Here is a standard plotting procedure (with few points since $H(x)$ is a simple function):

```
x = linspace(-10, 10, 5)
y = H(x)
plot(x, y)
```

- First problem: `ValueError` error in `H(x)` from `if x < 0`
- Let us debug in an interactive shell:

```
>>> x = linspace(-10,10,5)
>>> x
array([-10.,  -5.,   0.,   5.,  10.])
>>> b = x < 0
>>> b
array([ True,  True, False, False, False], dtype=bool)
>>> bool(b)  # evaluate b in a boolean context
...
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```


Plotting the Heaviside function (part 3)

- There are two remedies:
  - make a loop over `x` values (simple)
  - use a tool to automatically vectorize `H(x)` (simple)
  - code the `if` test in another way (most efficient)
- We look at the loop version first:

```
def H_loop(x):
    r = zeros(len(x))  # or r = x.copy()
    for i in xrange(len(x)):
        r[i] = H(x[i])
    return r

x = linspace(-5, 5, 6)
y = H_loop(x)
```


Plotting the Heaviside function (part 4)

- Automatic vectorization (no loops):

```
from numpy import vectorize
Hv = vectorize(H)
# Hv(x) works with array x
```

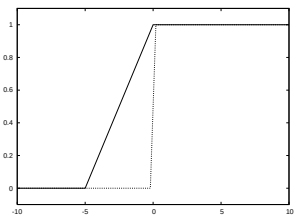
  but the resulting function is as slow as explicit loops
- Best (and most advanced) method: vectorize the `if` test

```
def f(x):
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x

def f_vectorized(x):
    return where(condition, expr1, expr2)
    # result[i] = expr1 if condition[i] else expr2

def Hv(x):
    return where(x < 0, 0.0, 1.0)
```


Plotting the Heaviside function (part 5)

- Back to plotting:

```
x = linspace(-10, 10, 5)   # linspace(-10, 10, 50)
plot(x, Hv(x), axis=[x[0], x[-1], -0.1, 1.1])
```



Plotting the Heaviside function (part 6)

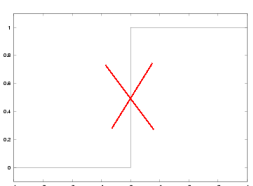
- We can add more and more x points, and the curve gets steeper and steeper
- Simpler strategy: plot two horizontal line segments
- One from $x = -10$ to $x = 0$, $y = 0$; and one from $x = 0$ to $x = 10$, $y = 1$

```
plot([-10, 0, 0, 10], [0, 0, 1, 1],
     axis=[x[0], x[-1], -0.1, 1.1])
```

- Remember: `plot(x,y)` just means drawing straight lines between `(x[0],y[0]), (x[1],y[1]), ...`

Plotting the Heaviside function (part 7)

- Some will argue and say that at high school they would draw $H(x)$ as two horizontal lines *without* the vertical line at $x = 0$ (illustrating the jump)
- How can we plot such a curve? (Plot two separate curves)

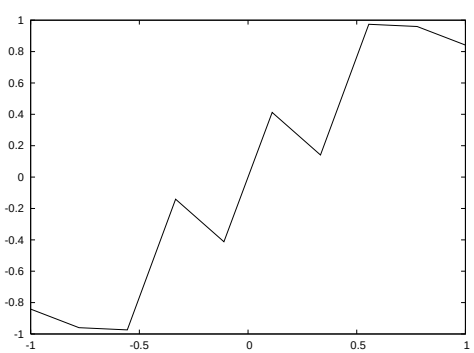


Plotting a rapidly varying function; code

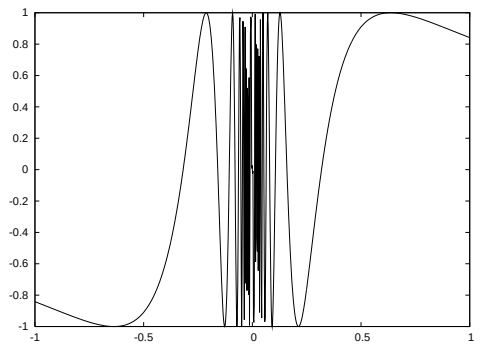
- Consider plotting $f(x) = \sin(1/x)$

```
def f(x):
    return sin(1.0/x)

x1 = linspace(-1, 1, 10)    # use 10 points
x2 = linspace(-1, 1, 1000)  # use 1000 points
plot(x1, f(x1), label='%d points' % len(x))
plot(x2, f(x2), label='%d points' % len(x))
```


Plotting a rapidly varying function; 10 points



Plotting a rapidly varying function; 1000 points



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.])
```


Two-dimensional arrays; math intro

- When we have a table of numbers,

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

(called *matrix* by mathematicians) it is natural to use a two-dimensional array $A_{i,j}$ with one index for the rows and one for the columns:

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Two-dimensional arrays; Python code

- Making and filling a two-dimensional NumPy array goes like this:

```
A = zeros((3,4))   # 3x4 table of numbers
A[0,0] = -1
A[1,0] =  1
A[2,0] = 10
A[0,1] = -5
...
A[2,3] = -100

# can also write (as for nested lists)
A[2][3] = -100
```


From nested list to two-dimensional array

- Let us make a table of numbers in a nested list:

```
>>> Cdegrees = [-30 + i*10 for i in range(3)]
>>> Fdegrees = [9./5*C + 32 for C in Cdegrees]
>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
>>> print table
[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
```

- Turn into NumPy array:

```
>>> table2 = array(table)
>>> print table2
[[-30. -22.]
 [-20.  -4.]
 [-10.  14.]]
```


Operations on two-dimensional arrays

- To see the number of elements in each dimension:

```
>>> table2.shape
(3, 2)              # 3 rows, 2 columns
```

- A for loop over all array elements:

```
>>> for i in range(table2.shape[0]):
...     for j in range(table2.shape[1]):
...         print 'table2[%d,%d] = %g' % (i, j, table2[i,j])
...
table2[0,0] = -30
table2[0,1] = -22
...
table2[2,1] = 14
```

- Alternative (single) loop over all elements:

```
>>> for index_tuple, value in ndenumerate(table2):
...     print 'index %s has value %g' % \
...           (index_tuple, table2[index_tuple])
...
index (0,0) has value -30
index (0,1) has value -22
...
index (2,1) has value 14
>>> type(index_tuple)
<type 'tuple'>
```


Slices of two-dimensional arrays (part 1)

- Rule: can use slices `start:stop:inc` for each index
- Extract the second column:

```
table2[0:table2.shape[0], 1]  # 2nd column (index 1)
array([-22.,  -4.,  14.])

>>> table2[0:, 1]             # same
array([-22.,  -4.,  14.])

>>> table2[:, 1]              # same
array([-22.,  -4.,  14.])
```


Slices of two-dimensional arrays (part 2)

- More slicing, with a bigger array:

```
>>> t = linspace(1, 30, 30).reshape(5, 6)
>>> t[1:-1:2, 2:]
array([[  9.,  10.,  11.,  12.],
       [ 21.,  22.,  23.,  24.]])
>>> t
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])
```

- What will `t[1:-1:2, 2:]` be?
- Slice `1:-1:2` for first index results in

```
[  7.,   8.,   9.,  10.,  11.,  12.]
[ 19.,  20.,  21.,  22.,  23.,  24.]
```

- Slice `2:` for the second index then gives

```
[  9.,  10.,  11.,  12.]
[ 21.,  22.,  23.,  24.]
```


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)
```


Summary of plotting $y = f(x)$ curves

- Curve plotting:

```
from scitools.std import *

plot(x, y)          # simplest command

plot(t1, y1, 'r',  # curve 1, red line
     t2, y2, 'b',  # curve 2, blue line
     t3, y3, 'o',  # curve 3, circles at data points
     axis=[t1[0], t1[-1], -1.1, 1.1],
     legend=('model 1', 'model 2', 'measurements'),
     xlabel='time', ylabel='force',
     hardcopy='myframe_%04d.png' % plot_counter)
```

- Straight lines are drawn between each data point
- Movies: make a hardcopy for each frame, then combine frames

```
movie('myframes_*.png', encoder='convert',
      output_file='movie.gif', fps=2)
```


Array functionality

`array(ld)`	copy list data `ld` to a `numpy` array
`asarray(d)`	make array of data `d` (copy if necess
`zeros(n)`	make a vector/array of length `n`, with
`zeros(n, int)`	make a vector/array of length `n`, with
`zeros((m,n), float)`	make a two-dimensional with shape
`zeros(x.shape, x.dtype)`	make array with shape and element
`linspace(a,b,m)`	uniform sequence of `m` numbers betv
`seq(a,b,h)`	uniform sequence of numbers from a
`iseq(a,b,h)`	uniform sequence of integers from a
`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 a

Summarizing example: animating a function (part 1)

- Goal: visualize the temperature in the ground as a function of depth (z) and time (t), displayed as a movie in time:

$$T(z,t) = T_0 + Ae^{-az}\cos(\omega t - az), \quad a = \sqrt{\frac{\omega}{2k}}$$

- First we make a *general* animation function for an $f(x,t)$:

```
def animate(tmax, dt, x, function, ymin, ymax, t0=0,
            xlabel='x', ylabel='y', hardcopy_stem='tmp_'):
    t = t0
    counter = 0
    while t <= tmax:
        y = function(x, t)
        plot(x, y,
             axis=[x[0], x[-1], ymin, ymax],
             title='time=%g' % t,
             xlabel=xlabel, ylabel=ylabel,
             hardcopy=hardcopy_stem + '%04d.png' % counter)
        t += dt
        counter += 1
```

- Then we call this function with our special $T(z,t)$ function

Summarizing example: animating a function (part 2)

```
# remove old plot files:
import glob, os
for filename in glob.glob('tmp_*.png'): os.remove(filename)

def T(z, t):
    # T0, A, k, and omega are global variables
    a = sqrt(omega/(2*k))
    return T0 + A*exp(-a*z)*cos(omega*t - a*z)

k = 1E-6     # heat conduction coefficient (in m*m/s)
P = 24*60*60.# oscillation period of 24 h (in seconds)
omega = 2*pi/P
dt = P/24    # time lag: 1 h
tmax = 3*P   # 3 day/night simulation
T0 = 10      # mean surface temperature in Celsius
A = 10       # amplitude of the temperature variations (in C)
a = sqrt(omega/(2*k))
D = -(1/a)*log(0.001) # max depth
n = 501      # no of points in the z direction

z = linspace(0, D, n)
animate(tmax, dt, z, T, T0-A, T0+A, 0, 'z', 'T')
# make movie files:
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')
```