Python programming — numeric Python

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

September 10, 2013

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]          # Not [3, 6, 9] !
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]          # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                    # Wants [2, 3, 4] ...
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]          # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                    # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]          # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                    # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]   # Matrix multiplication
```

Python numerics

The problem with Python:

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]          # Not [3, 6, 9] !

>>> [1, 2, 3] + 1                    # Wants [2, 3, 4] ...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list

>>> [[1, 2], [3, 4]] * [[5, 6], [7, 8]]   # Matrix multiplication
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

`map()`, `reduce()` and `filter()` ...

Poor-man's vector operations: the `map()` built-in function:

```
>>> map(lambda x : 3*x, [1, 2, 3])
[3, 6, 9]
```

`lambda` is for an anonymous function, `x` the input argument, and `3*x` the function and the return argument. Also possible with ordinary functions:

```
>>> from math import sin, pow
>>> map(sin, [1, 2, 3])
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]

>>> map(pow, [1, 2, 3], [3, 2, 1])
[1.0, 4.0, 3.0]
```

...`map()`, `reduce()` and `filter()`

`reduce()` collapses a list

```
>>> def mysum(x,y):
...     return x+y
...
>>> reduce(mysum, [1, 2, 3, 4, 5])
15
```

The results corresponds to `(((1+2)+3)+4)+5`

`filter()` takes out individual elements from the list:

```
>>> filter(lambda x: mod(x,2) == 0, [1, 2, 3, 4, 5])
[2, 4]
```

The function in the first argument should return `True` or `False`.

Solution: NumPy

```
>>> from numpy import *
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...

```
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
```

Solution: NumPy

```
>>> from numpy import *
>>> array([1, 2, 3]) * 3
array([3, 6, 9])
>>> array([1, 2, 3]) + 1
array([2, 3, 4])
>>> array([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
array([[ 5, 12],
       [21, 32]])
```

This is elementwise multiplication...

```
>>> matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
matrix([[19, 22],
        [43, 50]])
```

Numerical and scientific Python

http://wiki.python.org/moin/NumericAndScientific

NumPy: Adds array (including matrix) functionality to Python

SciPy: Linear algebra, signal and image processing, statistics, etc.

Other: PyGSL, ScientificPython, PyChem (Jarvis et al., 2006) for multivariate chemometric, . . .

Visualizaton: **matplotlib**, VTK, PIL, Gnuplot, . . .

History: Earlier version of `NumPy` was called `Numeric` and `numarray` and older books such as (Langtangen, 2005) and webpages may describe these modules.

IPython

"An Enhanced Interactive Python" with automatic completion and interactive plotting if called with `pylab`:

```
$ ipython -pylab

In [1]: matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]]
Out[1]:
matrix([[19, 22],
        [43, 50]])

In [2]: plot(matrix([[1, 2], [3, 4]]) * [[5, 6], [7, 8]])
```

Numpy arrays ...

There are two basic types `array` and `matrix`:

An `array` may be a vector (one-dimensional array)

```
>>> from numpy import *
>>> array([1, 2, 3, 4])
array([1, 2, 3, 4])
```

Or a matrix (a two-dimensional array)

```
>>> array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

. . . Numpy arrays . . .

Or a higher-order tensor. Here a 2-by-2-by-2 tensor:

```
>>> array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

N, $1 \times N$ and $N \times 1$ `array`-s are different:

```
>>> array([[1, 2, 3, 4]])[2]              # There is no third row
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index out of bounds
```

. . . Numpy arrays

A `matrix` is always two-dimensional, e.g., this

```
>>> matrix([1, 2, 3, 4])
matrix([[1, 2, 3, 4]])
```

is a two-dimensional data structure with one row and four columns

. . . and with a 2-by-2 matrix:

```
>>> matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

Numpy arrays copy

To copy by reference use `asmatrix()` or `mat()` (from an `array`) or `asarray()` (from a `matrix`)

```
>>> a = array([1, 2, 3, 4])
>>> m = asmatrix(a)              # Copy as reference
>>> a[0] = 1000
>>> m
matrix([[1000,    2,    3,    4]])
```

To copy elements use `matrix()` or `array()`

```
>>> a = array([1, 2, 3, 4])
>>> m = matrix(a)                # Copy elements
>>> a[0] = 1000
>>> m
matrix([[1, 2, 3, 4]])
```

Datatypes

Elements are 4 bytes integers or 8 bytes float per default:

```
>>> array([1, 2, 3, 4]).itemsize    # Number of bytes for each element
4
>>> array([1., 2., 3., 4.]).itemsize
8
>>> array([1., 2, 3, 4]).itemsize   # Not heterogeneous
8
```

`array` and `matrix` can be called with datatype to set it otherwise:

```
>>> array([1, 2], 'int8').itemsize
1
>>> array([1, 2], 'float32').itemsize
4
```

Initialization of arrays

Functions `ones()`, `zeros()`, `eye()` (also `identity()`), `linspace()` work "as expected" (from Matlab), though the first argument for `ones()` and `zeros()` should contain the size in a list or tuple:

```
>>> zeros((1, 2))                    # zeros(1, 2) doesn't work
array([[ 0.,  0.]])
```

In Matlab it is possible to construct a vector with "`1:10`". In Python this is not possible, instead use:

```
>>> r_[1:11]
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> arange(1, 11)
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Diagonal . . .

The `diagonal()` function works both for `matrix` and two-dimensional `array`:

```
>>> diagonal(matrix([[1, 2], [3, 4]]))
array([1, 4])
```

Note now the vector/matrix is an one-dimensional `array`

It "works" for higher order `arrays`:

```
>>> diagonal(array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]))
array([[1, 7],
       [2, 8]])
```

`diagonal()` does not work for one-dimensional arrays.

. . . Diagonal

Yet another function: `diag()`. Works for one- and two-dimensional `matrix` and `array`

```
>>> m = matrix([[1, 2], [3, 4]])
>>> d = diag(m)
>>> d
array([1, 4])
>>> diag(d)
array([[1, 0],
       [0, 4]])
```

Like Matlab's `diag()`.

It is also possible to specify the diagonal: `diag(m,1)`

Matrix transpose

Matrix transpose is different with Python's `array` and `matrix` and Matlab

```
>>> A = array([[1+1j, 1+2j], [2+1j, 2+2j]]); A
array([[ 1.+1.j,  1.+2.j],
       [ 2.+1.j,  2.+2.j]])
```

`.T` for `array` and `matrix` (like Matlab "`.`") and `.H` for `matrix` (like Matlab "`,`"):

```
>>> A.T         # No conjucation. Also: A.transpose() or transpose(A)
array([[ 1.+1.j,  2.+1.j],
       [ 1.+2.j,  2.+2.j]])

>>> mat(A).H   # Complex conjugate transpose. Or: A.conj().T
matrix([[ 1.-1.j,  2.-1.j],
        [ 1.-2.j,  2.-2.j]])
```

Sizes and reshaping

A 2-by-2-by-2 tensor:

```
>>> a = array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> a.ndim                          # Number of dimensions
3

>>> a.shape                         # Number of elements in each dimension
(2, 2, 2)
>>> a.shape = (1,8); a              # Reshaping: 1-by-8 matrix
array([[1, 2, 3, 4, 5, 6, 7, 8]])
>>> a.shape = 8; a                  # 8-element vector
array([1, 2, 3, 4, 5, 6, 7, 8])
```

There is a related function for the last line:

```
>>> a.flatten()                     # Always copy
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1,1]
```

What happens here?

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1,1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been `L[1][1]`. With matrices:

```
>>> mat(L)[1,1]
```

Indexing . . .

Ordinary lists:

```
>>> L = [[1, 2], [3, 4]]
>>> L[1,1]
```

What happens here?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list indices must be integers
```

Should have been `L[1][1]`. With matrices:

```
>>> mat(L)[1,1]
4
```

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here?

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here? Error message:

```
IndexError: index out of bounds
```

```
>>> mat(L)[1]
```

. . . Indexing . . .

```
>>> mat(L)[1][1]
```

What happens here? Error message:

```
IndexError: index out of bounds
```

```
>>> mat(L)[1]
matrix([[3, 4]])
>>> asarray(L)[1]
array([3, 4])
>>> asarray(L)[1][1]
4
>>> asarray(L)[1,1]
4
```

Indexing with multiple indices

```
>>> A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> i = [1, 2]                  # Second and third row/column
>>> A[i,i]
matrix([[5, 9]])                # Elements from diagonal
>>> A[ix_(i,i)]                 # Block matrix
matrix([[5, 6],
        [8, 9]])
```

Take out third, “fourth” and “fifth” column with wrapping:

```
>>> A.take([2, 3, 4], axis=1, mode='wrap')
matrix([[3, 1, 2],
        [6, 4, 5],
        [9, 7, 8]])
```

Conditionals & concatenation

Construct a matrix based on a condition (first input argument)

```
>>> where(mod(A.flatten(), 2), 1, 0)     # Find odd numbers
matrix([[1, 0, 1, 0, 1, 0, 1, 0, 1]])
```

Horizontal concatenate as in Matlab "`[ A A ]`" and note tuple input!

```
>>> B = concatenate((A, A), axis=1)  # Or:
>>> B = bmat('A A')                  # Note string input. Or:
>>> hstack((A, A))
matrix([[1, 2, 3, 1, 2, 3],
        [4, 5, 6, 4, 5, 6],
        [7, 8, 9, 7, 8, 9]])
```

For concatenating rows use `vstack()`, `bmat('A ; A')` or `concatenate()`

Random initialization with `random`

Python with one element at a time with the `random` module:

```
>>> import random
>>> random.random()    # Float between 0 and 1
0.5466909536294228 8
>>> random.gauss(mu = 10, sigma = 3)
7.702673969795700 5
```

Other probability functions: beta, exponential, gamma, lognormal, Pareto, `randint`, `randrange`, `uniform`, Von Mises and Weibull.

```
>>> a = [1, 2, 3, 4]; random.shuffle(a); a
[3, 1, 4, 2]
```

Other functions `choice`, `sample`, etc.

Random initialization with `numpy`

Initialize an array with random numbers between zero and one:

```
>>> import numpy.random
>>> numpy.random.random((2,3))                    # Tuple input!
array([[ 0.98872329,  0.73451282,  0.54337299],
       [ 0.69088015,  0.59413038,  0.71935909]])
```

Standard Gaussian (normal distribution):

```
>>> numpy.random.randn(2, 3)                      # Individual input arg.
array([[-0.19301411, -1.37092092, -0.1666896 ],
       [ 1.41485887,  2.24646526, -1.27417696]])
>>> N = numpy.random.standard_normal((2, 3))    # Tuple input!
>>> N = numpy.random.normal(0, 1, (2, 3))
```

Random initialization with `scipy.stats`

10 discrete distributions and 81 continuous distributions

```
>>> scipy.stats.uniform.rvs(size=(2, 3))      # Uniform
array([[ 0.23273417,  0.17636535,  0.88709937],
       [ 0.07573364,  0.04084195,  0.45961136]])
>>> scipy.stats.norm.rvs(size=(2, 3))         # Gaussian
array([[ 0.89339055, -0.05093851,  0.12449392],
       [ 0.49639535, -1.39487053,  0.38580828]])
```

Multiplications and divisions . . .

Numpy multiplication and divisions are confusing.

```
>>> from numpy import *
>>> A = array([[1, 2], [3, 4]])
```

With `array` the "default" is elementwise:

```
>>> A * A                 # '*' is elementwise, Matlab: A.*A
array([[ 1,  4],
       [ 9, 16]])
>>> dot(A, A)             # dot() is matrix multiplication
array([[ 7, 10],
       [15, 22]])
```

. . . Multiplications and divisions

With `matrix` the default is matrix multiplication

```
>>> mat(A) * mat(A)           # Here '*' is matrix multiplication!
matrix([[ 7, 10],
        [15, 22]])
>>> multiply(mat(A), mat(A)) # 'multiply' is elementwise multiplication
matrix([[ 1,  4],
        [ 9, 16]])
>>> dot(mat(A), mat(A))       # dot() is matrix multiplication
matrix([[ 7, 10],
        [15, 22]])

>>> mat(A) / mat(A)           # Division always elementwise
matrix([[1, 1],
        [1, 1]])
```

Matrix inversion

"Ordinary" matrix inversion available as `inv()` in the `linalg` module of `numpy`

```
>>> linalg.inv(mat([[2, 1], [1, 2]]))
matrix([[ 0.66666667, -0.33333333],
        [-0.33333333,  0.66666667]])
```

Pseudo-inverse `linalg.pinv()` for singular matrices:

```
>>> linalg.pinv(mat([[2, 0], [0, 0]]))
matrix([[ 0.5,  0. ],
        [ 0. ,  0. ]])
```

Singular value decomposition

`svd()` function in the `linalg` module returns by default three argument:

```
>>> from numpy.linalg import svd
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]))
```

gives a 2-by-2 matrix, a 2-vector with singular values and a 3-by-3 matrix:

```
>>> U * diag(s) * V       # Gives 'ValueError: matrices are not aligned'
>>> U, s, V = svd(mat([[1, 0, 0], [0, 0, 0]]), full_matrices=False)
>>> U * diag(s) * V       # Now ok: V.shape == (2, 3)
```

Note $\mathbf{V}$ is transposed compared to Matlab!

If only the singular values are required use `compute_uv` argument:

```
>>> s = svd(mat([[1, 0, 0], [0, 0, 0]]), compute_uv=False)
```

Non-negative matrix factorization . . .

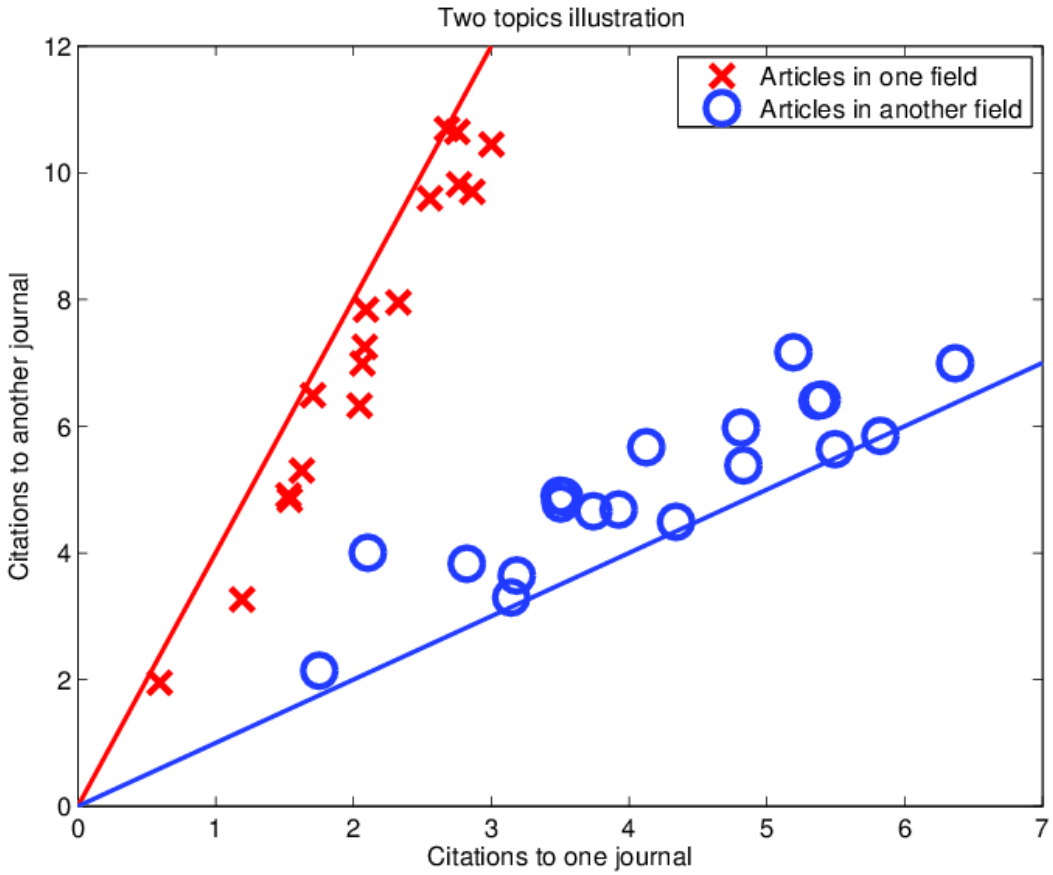


Figure 1: Illustration of non-negative matrix factorization with made up citation data and where the lines represents the loadings in the $\mathbf{H}$ matrix.

Factorization: $\mathbf{X} = \mathbf{WH}$ (Lee and Seung, 2001; Segaran, 2007)

Compared to singular value decomposition ("latent semantic indexing"):

— Clusters are not constraint to be orthogonal ☺

— Only one "interpretable" end for NMF ☺

— Seemingly more difficult estimation ☹

. . . Non-negative matrix factorization

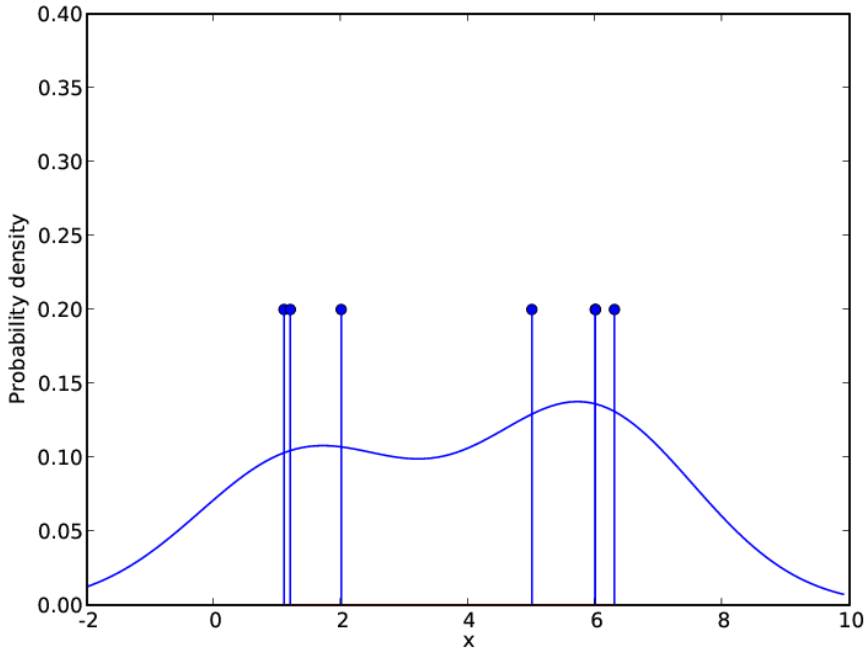
```
def nmf(M, components=5, iterations=5000):
    """ non-negative matrix factorization. Returns two factorized
        matrices
    """
    from numpy import mat, random, multiply
    W = mat(random.rand(M.shape[0], components))
    H = mat(random.rand(components, M.shape[1]))
    for n in range(0, iterations):
        H = multiply(H, (W.T * M) / (W.T * W * H + 0.001))
        W = multiply(W, (M * H.T) / (W * (H * H.T) + 0.001))
        print "%d/%d" % (n, iterations)
    return (W, H)
```

Note “0.001” needs to be set to some appropriate for the dataset.

Kernel density estimation . . .

```
from pylab import *; import scipy.stats

x0 = array([1.1, 1.2, 2, 6, 6,5, 6.3])
x  = arange(-3, 10, 0.1)
plot(x, scipy.stats.gaussian_kde(x0).evaluate(x))
```

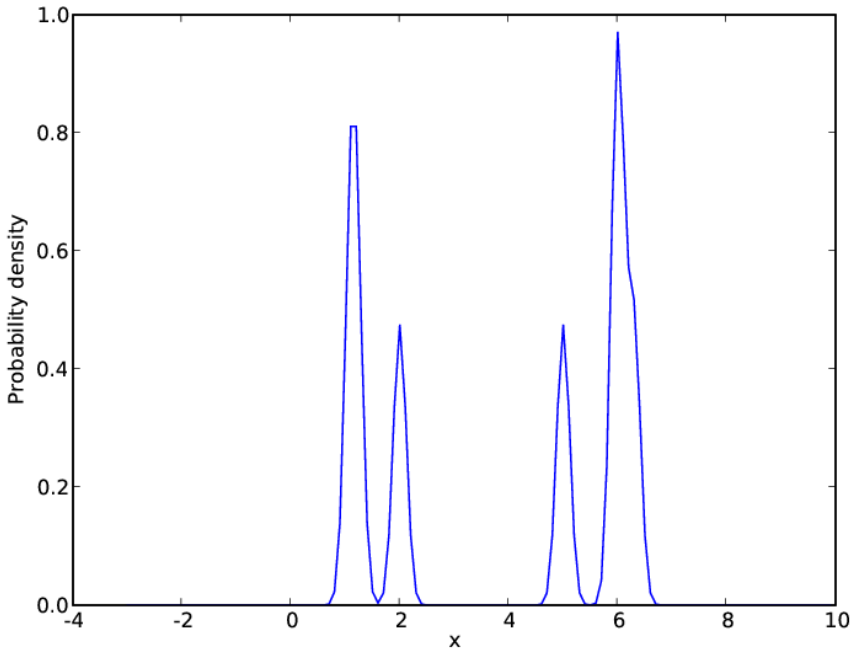


Bandwidth computed via "scott's factor" . . . ?

Estimation in higher dimensions possible

. . . Kernel density estimation

```
>>> class kde(scipy.stats.gaussian_kde):
...   def covariance_factor(dummy): return 0.05
...
>>> plot(x, kde(x0).evaluate(x))
```



Defining a new class "`kde`" from the `gaussian_kde` class from the `scipy.stats.gaussian_kde` module with

Fourier transformation

```
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

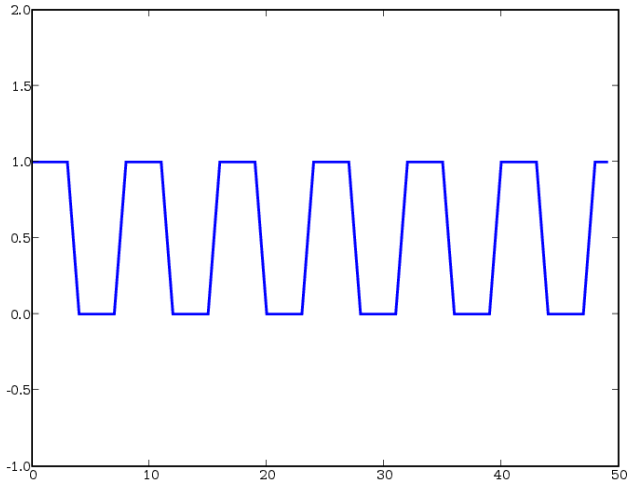
What is `x`?

Fourier transformation . . .

```
>>> x = asarray([[1] * 4, [0] * 4] * 100).flatten()
```

What is `x`?

```
>>> x[:20]
array([1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1])
```

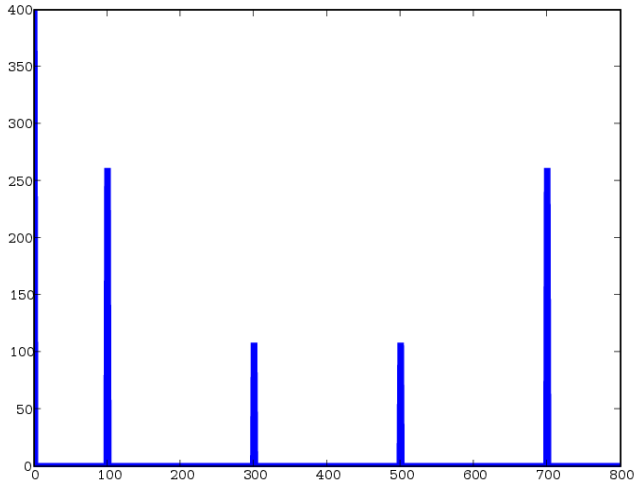


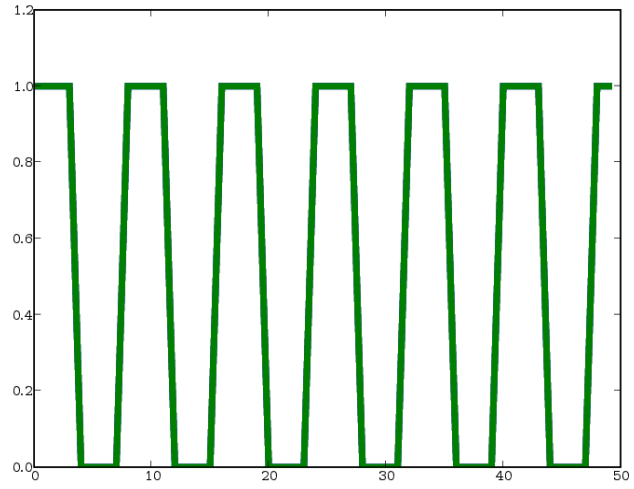
Zoom in on the first 51 data points of the square wave signal

With the `pylab` module:

```
>>> plot(x[:50], linewidth=2)
>>> axis((0, 50, -1, 2))
```

. . . Fourier transformation





Forward Fourier transformation with the `fft()` from `numpy.fft.fftpack` module (also loaded with `scipy`, `numpy.fft`, `pylab` so look out for naming conflict):

```
>>> abs(fft(x))
```

Back to "time" domain with inverse Fourier transformation:

```
>>> from numpy.fft import *
>>> abs(ifft(fft.fft(x)))[:6].round()
array([ 1.,  1.,  1.,  1.,  0.,  0.])
```

Other: `fft2()`, `fftn()`

Optimization with `scipy.optimize`

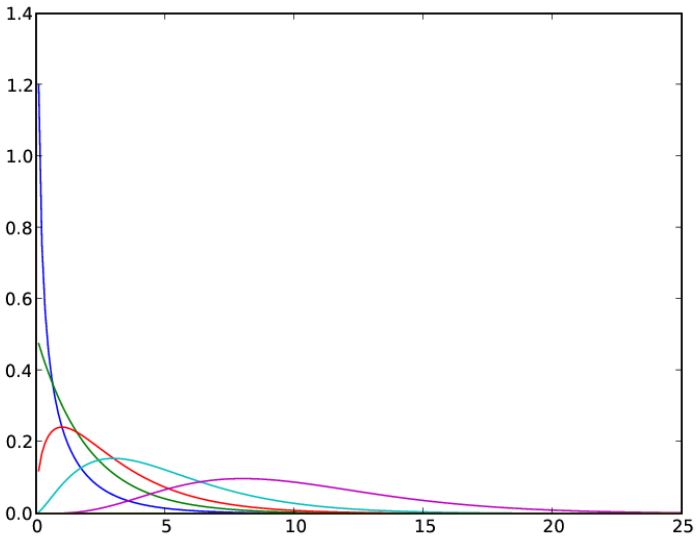
`scipy.optimize` contains functions for mathematical function optimization:

`fmin()` is Nelder-Mead simplex algorithm for function optimization without derivatives.

```
>>> import scipy.optimize, math
>>> scipy.optimize.fmin(math.cos, [1])
Optimization terminated successfully.
         Current function value: -1.000000
         Iterations: 19
         Function evaluations: 38
array([ 3.14160156])
```

Other optimizers: `fmin_cg()`, `leastsq()`, `...`

Statistics with `scipy.stats`



Showing the χ^2 probablity density function with different degrees of freedom:

```
from pylab import *
from scipy.stats import chi2
x = linspace(0.1, 25, 200)
for dof in [1, 2, 3, 5, 10, 50]:
    plot(x, chi2.pdf(x, dof))
```

Other functions such as missing data mean:

```
>>> x = [1, nan, 3, 4, 5, 6, 7, 8, nan, 10]
>>> scipy.stats.stats.nanmean(x)
5.5
```

Reading a matrix from files

Wikipedia citation data from (Nielsen, 2008)

```
f = open('enwiki-20080312-ref-articlejournal_rows.txt')
rows = [ line.strip() for line in f ]

f = open('enwiki-20080312-ref-articlejournal_columns.txt')
columns = [ line.strip() for line in f ]

M = mat(zeros((len(rows), len(columns)), 'uint8'))
f = open('enwiki-20080312-ref-articlejournal_indices.txt')
for line in f:
    [i, j, value] = map(int, line.split())
    M[i-1,j-1] = min(value, 255)
```

Sparse matrices: `scipy.sparse`

```
f = open('enwiki-20080312-ref-articlejournal_rows.txt')
rows = [ line.strip() for line in f ]

f = open('enwiki-20080312-ref-articlejournal_columns.txt')
columns = [ line.strip() for line in f ]



from scipy import sparse
M = sparse.lil_matrix((len(rows), len(columns)))
f = open('enwiki-20080312-ref-articlejournal_indices.txt')
for line in f:
    [i, j, value] = map(int, line.split())
    M[i-1,j-1] = value
```

NMF with sparse data matrix

```
def nmf_sparse(M, components=5, iterations=5000):
    import numpy
    W = numpy.mat(numpy.random.rand(M.shape[0], components))
    H = numpy.mat(numpy.random.rand(components, M.shape[1]))
    for n in range(0, iterations):
        H = numpy.multiply(H, (W.T * M).todense() / (W.T * W * H + 0.001))
        W = numpy.multiply(W, (M * H.T).todense() / (W * (H * H.T) + 0.001)
        print "%d/%d" % (n, iterations)
    return (W, H)
```

Sparse matrices: pysparse

```
f = open('enwiki-20080312-ref-articlejournal_rows.txt')
rows = [ line.strip() for line in f ]

f = open('enwiki-20080312-ref-articlejournal_columns.txt')
columns = [ line.strip() for line in f ]

e = len([line for line in open('enwiki-20080312-ref-articlejournal_indices.

from pysparse import spmatrix
M = spmatrix.ll_mat(len(rows), len(columns), e)
f = open('enwiki-20080312-ref-articlejournal_indices.txt')
for line in f:
    [i, j, value] = map(int, line.split())
    M[i-1,j-1] = value
M = M.to_csr()
```

Structured matrices

pandas: Python Data Analysis Library. New data analysis package for Python with DataFrame like *R*.

Python `MetaArray`, http://www.scipy.org/Cookbook/MetaArray : An N-way annotated array

In the Brede Toolbox (Nielsen and Hansen, 2000) I store data in Matlab "structures", — somewhat similar to Python's dictionaries

```
          rows: {1x186 cell}
       columns: {1x789 cell}
        matrix: [186x789 double]
   description: 'Author.keyname [ X(186 x 789) ]'
          type: 'mat'
```

Example on reading stock quotes . . .

Getting stock data for Novo Nordisk (Segaran, 2007, p. 244+)

```
import urllib2, numpy
url = ("http://ichart.finance.yahoo.com/table.csv?"
       "s=NVO&d=11&e=26&f=2006&g=d&a=3&b=12&c=1999&ignore=.csv")
rows = urllib2.urlopen(url).readlines()
```

Content of the downloaded file:

```
>>> rows[0]                          # Column header
'Date,Open,High,Low,Close,Volume,Adj Close\n'
>>> rows[1:3]                        # First two content lines
['2006-12-26,84.60,84.65,84.01,84.29,53000,40.11\n',
'2006-12-22,84.32,84.55,83.90,84.35,87800,40.14\n']
```

. . . Example on reading stock quotes

```
M = {}
M["columns"] = rows[0].strip().split(",")[1:]
M["rows"]    = [ row.split(",")[0] for row in rows[1:] ]
M["matrix"]  = matrix([[ float(e) for e in row.strip().split(",")[1:]]
                                  for row in rows[1:]])
```

In Pandas it is very simple as it handles csv even referenced with URLs:

```
>>> import pandas as pd
>>> stock = pd.read_csv(url, index_col=0)
>>> stock[0:2]
             Open   High    Low  Close  Volume  Adj Close
Date
2006-12-26  84.60  84.65  84.01  84.29   53000      37.42
2006-12-22  84.32  84.55  83.90  84.35   87800      37.45
```

Profiling

`time` and `timeit` modules and `times()` function in the `os` module (Langtangen, 2005, p. 422+):

```
from numpy import array, dot
import time

elapsed = time.time()
cpu = time.clock()
dummy = dot(array((100000,1)), array((100000,1)))
print time.time() - elapsed
print time.clock() - cpu
```

Profiling: `profile`, `hotshot` and `pstat`.

Six advices from Hans Petter Langtangen

Section *Optimization of Python code* (Langtangen, 2005, p. 426+)

Avoid loops, use NumPy (see also my blog)

Avoid prefix in often called functions.

Plain functions run faster than class methods

Don't use NumPy for scalar arguments

Use `xrange` instead of `range` (in Python < 3)

`if-else` is faster than `try-except`

Interface to Matlab & R

`pymat`, interface to Matlab

Matlab to Python compiler (Jurica and van Leeuwen, 2009), http://ompc.juricap.com

R interface, `rpy` and `rpy2`, http://rpy.sourceforge.net/

```
>>> import rpy
>>> rpy.r.wilcox_test([1,2,3], [4,5,6])
{'null.value': {'location shift': 0.0}, 'data.name': '1:3 and 4:6',
'p.value': 0.10000000000000001, 'statistic': {'W': 0.0},
'alternative': 'two.sided', 'parameter': None,
'method': 'Wilcoxon rank sum test'}
```

More information

http://www.scipy.org/NumPy_for_Matlab_Users

MATLAB commands in numerical Python (Numpy), Vidar Bronken Gundersen, mathesaurus.sf.net

Guide to NumPy (Oliphant, 2006)

Videolectures.net: John D. Hunter overview of Matplotlib
http://videolectures.net/mloss08_hunter_mat/

References

Jarvis, R. M., Broadhurst, D., Johnson, H., O'Boyle, N. M., and Goodacre, R. (2006). PYCHEM: a multivariate analysis package for python. *Bioinformatics*, 22(20):2565–2566. DOI: 10.1093/bioinformatics/btl416.

Jurica, P. and van Leeuwen, C. (2009). OMPC: an open-source MATLAB(R)-to-Python compiler. *Frontiers in Neuroinformatics*, 3:5. DOI: 10.3389/neuro.11.005.2009.

Langtangen, H. P. (2005). *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer. ISBN 3540294155.

Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, pages 556–562, Cambridge, Massachusetts. MIT Press. http://hebb.mit.edu/people/seung/papers/nmfconverge.pdf. CiteSeer: http://citeseer.ist.psu.edu/-lee00algorithms.html.

Nielsen, F. Å. (2008). Clustering of scientific citations in Wikipedia. In *Wikimania*. http://arxiv.org/abs/0805.1154.

Nielsen, F. Å. and Hansen, L. K. (2000). Experiences with Matlab and VRML in functional neuroimaging visualizations. In Klasky, S. and Thorpe, S., editors, *VDE2000 - Visualization Development Environments, Workshop Proceedings, Princeton, New Jersey, USA, April 27–28, 2000*, pages 76–81, Princeton, New Jersey. Princeton Plasma Physics Laboratory. http://www.imm.dtu.dk/pubdb/views/edoc_download.php-/1231/pdf/imm1231.pdf. CiteSeer: http://citeseer.ist.psu.edu/309470.html.

Oliphant, T. E. (2006). *Guide to NumPy*. Trelgol Publishing.

Segaran, T. (2007). *Programming Collective Intelligence*. O'Reilly, Sebastopol, California.