NUMPY FOR MATLAB USERS

5.1 Introduction

MATLAB® and NumPy/SciPy have a lot in common. But there are many differences. NumPy and SciPy were created to do numerical and scientific computing in the most natural way with Python, not to be MATLAB® clones. This page is intended to be a place to collect wisdom about the differences, mostly for the purpose of helping proficient MATLAB® users become proficient NumPy and SciPy users.

5.2 Some Key Differences

In MATLAB®, the basic data type is a multidimensional array of double precision floating point numbers. Most expressions take such arrays and return such arrays. Operations on the 2-D instances of these arrays are designed to act more or less like matrix operations in linear algebra.	In NumPy the basic type is a multidimensional `array`. Operations on these arrays in all dimensionalities including 2D are element-wise operations.
MATLAB® uses 1 (one) based indexing. The initial element of a sequence is found using a(1). *See note INDEXING*	Python uses 0 (zero) based indexing. The initial element of a sequence is found using a[0].
MATLAB®'s scripting language was created for doing linear algebra. The syntax for basic matrix operations is nice and clean, but the API for adding GUIs and making full-fledged applications is more or less an afterthought.	NumPy is based on Python, which was designed from the outset to be an excellent general-purpose programming language. While Matlab's syntax for some array manipulations is more compact than NumPy's, NumPy (by virtue of being an add-on to Python) can do many things that Matlab just cannot, for instance subclassing the main array type to do both array and matrix math cleanly.
In MATLAB®, arrays have pass-by-value semantics, with a lazy copy-on-write scheme to prevent actually creating copies until they are actually needed. Slice operations copy parts of the array.	In NumPy arrays have pass-by-reference semantics. Slice operations are views into an array.

- Operator `*`, `dot()`, and `multiply()`:
  - For `array`, **“‘*“‘ means element-wise multiplication**, and the `dot()` function is used for matrix multiplication.

- Handling of vectors (one-dimensional arrays)
  - For `array`, the **vector shapes 1xN, Nx1, and N are all different things**. Operations like `A[:,1]` return a one-dimensional array of shape N, not a two-dimensional array of shape Nx1. Transpose on a one-dimensional `array` does nothing.

- Handling of higher-dimensional arrays (ndim > 2)
  - `array` objects **can have number of dimensions > 2**;

- Convenience attributes
  - `array` **has a .T attribute**, which returns the transpose of the data.

- Convenience constructor
  - The `array` constructor **takes (nested) Python sequences as initializers**. As in, `array([[1,2,3], [4,5,6]])`.

There are pros and cons to using both:

- `array`
  - `:)` You can treat one-dimensional arrays as *either* row or column vectors. `dot(A,v)` treats `v` as a column vector, while `dot(v,A)` treats `v` as a row vector. This can save you having to type a lot of transposes.
  - `<:(` Having to use the `dot()` function for matrix-multiply is messy—`dot(dot(A,B),C)` vs. `A*B*C`. This isn't an issue with Python >= 3.5 because the `@` operator allows it to be written as `A @ B @ C`.
  - `:)` Element-wise multiplication is easy: `A*B`.
  - `:)` Closer in semantics to tensor algebra, if you are familiar with that.
  - `:)` *All* operations (`*`, `/`, `+`, `-` etc.) are element-wise.

In the table below, it is assumed that you have executed the following commands in Python:

```
from numpy import *
import scipy.linalg
```

Also assume below that if the Notes talk about "matrix" that the arguments are two-dimensional entities.

5.5.1 General Purpose Equivalents

MATLAB	numpy	Notes
`help func`	`info(func)` or `help(func)` or `func?` (in Ipython)	get help on the function *func*
`which func`	see note HELP	find out where *func* is defined
`type func`	`source(func)` or `func??` (in Ipython)	print source for *func* (if not a native function)
`a && b`	`a and b`	short-circuiting logical AND operator (Python native operator); scalar arguments only
`a || b`	`a or b`	short-circuiting logical OR operator (Python native operator); scalar arguments only
`1*i, 1*j, 1i, 1j`	`1j`	complex numbers
`eps`	`np.spacing(1)`	Distance between 1 and the nearest floating point number.
`ode45`	`scipy.integrate.solve_ivp(f)`	integrate an ODE with Runge-Kutta 4,5
`ode15s`	`scipy.integrate.solve_ivp(f, method='BDF')`	integrate an ODE with BDF method

5.5.2 Linear Algebra Equivalents

MATLAB	NumPy
`ndims(a)`	`ndim(a)` or `a.ndim`
`numel(a)`	`size(a)` or `a.size`
`size(a)`	`shape(a)` or `a.shape`
`size(a,n)`	`a.shape[n-1]`
`[ 1 2 3; 4 5 6 ]`	`array([[1.,2.,3.], [4.,5.,6.]])`
`[ a b; c d ]`	`vstack([hstack([a,b]), hstack([c,d])])` `bmat('a b; c d').A`

MATLAB	NumPy
a(end)	a[-1]
a(2,5)	a[1,4]
a(2,:)	a[1] or a[1,:]
a(1:5,:)	a[0:5] or a[:5] or a[0:5,:]
a(end-4:end,:)	a[-5:]
a(1:3,5:9)	a[0:3][:,4:9]
a([2,4,5],[1,3])	a[ix_([1,3,4],[0,2])]
a(3:2:21,:)	a[2:21:2,:]
a(1:2:end,:)	a[::2,:]
a(end:-1:1,:) or flipud(a)	a[::-1,:]
a([1:end 1],:)	a[r_[:len(a),0]]
a.'	a.transpose() or a.T
a'	a.conj().transpose() or a.conj().T
a * b	a.dot(b)
a .* b	a * b
a./b	a/b
a.^3	a**3
(a>0.5)	(a>0.5)
find(a>0.5)	nonzero(a>0.5)
a(:,find(v>0.5))	a[:,nonzero(v>0.5)[0]]
a(:,find(v>0.5))	a[:,v.T>0.5]
a(a<0.5)=0	a[a<0.5]=0
a .* (a>0.5)	a * (a>0.5)
a(:) = 3	a[:] = 3
y=x	y = x.copy()
y=x(2,:)	y = x[1,:].copy()
y=x(:)	y = x.flatten()
1:10	arange(1.,11.) or r_[1.:11.] or r_[1:10:10j]
0:9	arange(10.) or r_[:10.] or r_[:9:10j]
[1:10]'	arange(1.,11.)[:, newaxis]
zeros(3,4)	zeros((3,4))
zeros(3,4,5)	zeros((3,4,5))
ones(3,4)	ones((3,4))
eye(3)	eye(3)
diag(a)	diag(a)
diag(a,0)	diag(a,0)
rand(3,4)	random.rand(3,4)
linspace(1,3,4)	linspace(1,3,4)
[x,y]=meshgrid(0:8,0:5)	mgrid[0:9.,0:6.] or meshgrid(r_[0:9.],r_[0:6.] ogrid[0:9.,0:6.] or ix_(r_[0:9.],r_[0:6.]
[x,y]=meshgrid([1,2,4],[2,4,5])	meshgrid([1,2,4],[2,4,5]) ix_([1,2,4],[2,4,5])
repmat(a, m, n)	tile(a, (m, n))
[a b]	concatenate((a,b),1) or hstack((a,b)) or column_stack((a,b))
[a; b]	concatenate((a,b)) or vstack((a,b)) or r_[a,b]
max(max(a))	a.max()
max(a)	a.max(0)
max(a,[],2)	a.max(1)
max(a,b)	maximum(a, b)

MATLAB	NumPy
`norm(v)`	`sqrt(dot(v,v))` or `np.linalg.norm(v)`
`a & b`	`logical_and(a,b)`
`a | b`	`logical_or(a,b)`
`bitand(a,b)`	`a & b`
`bitor(a,b)`	`a | b`
`inv(a)`	`linalg.inv(a)`
`pinv(a)`	`linalg.pinv(a)`
`rank(a)`	`linalg.matrix_rank(a)`
`a\b`	`linalg.solve(a,b)` if `a` is square; `linalg.lstsq(a,b)` otherwise
`b/a`	Solve a.T x.T = b.T instead
`[U,S,V]=svd(a)`	`U, S, Vh = linalg.svd(a), V = Vh.T`
`chol(a)`	`linalg.cholesky(a).T`
`[V,D]=eig(a)`	`D,V = linalg.eig(a)`
`[V,D]=eig(a,b)`	`V,D = np.linalg.eig(a,b)`
`[Q,R,P]=qr(a,0)`	`Q,R = scipy.linalg.qr(a)`
`[L,U,P]=lu(a)`	`L,U = scipy.linalg.lu(a)` or `LU,P=scipy.linalg.lu_factor(a)`
`conjgrad`	`scipy.sparse.linalg.cg`
`fft(a)`	`fft(a)`
`ifft(a)`	`ifft(a)`
`sort(a)`	`sort(a)` or `a.sort()`
`[b,I] = sortrows(a,i)`	`I = argsort(a[:,i]), b=a[I,:]`
`regress(y,X)`	`linalg.lstsq(X,y)`
`decimate(x, q)`	`scipy.signal.resample(x, len(x)/q)`
`unique(a)`	`unique(a)`
`squeeze(a)`	`a.squeeze()`

5.6 Notes

Submatrix: Assignment to a submatrix can be done with lists of indexes using the `ix_` command. E.g., for 2d array `a`, one might do: `ind=[1,3]; a[np.ix_(ind,ind)]+=100`.

HELP: There is no direct equivalent of MATLAB's `which` command, but the commands `help` and `source` will usually list the filename where the function is located.

INDEXING: MATLAB® uses one based indexing, so the initial element of a sequence has index 1. Python uses zero based indexing, so the initial element of a sequence has index 0.

RANGES: In MATLAB®, `0:5` can be used as both a range literal and a 'slice' index (inside parentheses); however, in Python, constructs like `0:5` can *only* be used as a slice index (inside square brackets). Thus the somewhat quirky `r_` object was created to allow numpy to have a similarly terse range construction mechanism. Note that `r_` is not called like a function or a constructor, but rather *indexed* using square brackets, which allows the use of Python's slice syntax in the arguments.

LOGICOPS: & or | in NumPy is bitwise AND/OR, while in Matlab & and | are logical AND/OR. The difference should be clear to anyone with significant programming experience. The two can appear to work the same, but there are important differences. If you would have used Matlab's & or | operators, you should use the NumPy ufuncs logical_and/logical_or. The notable differences between Matlab's and NumPy's & and | operators are:

- Non-logical {0,1} inputs: NumPy's output is the bitwise AND of the inputs. Matlab treats any non-zero value as 1 and returns the logical AND. For example (3 & 4) in NumPy is 0, while in Matlab both 3 and 4 are considered logical true and (3 & 4) returns 1.
- Precedence: NumPy's & operator is higher precedence than logical operators like < and >; Matlab's is the reverse.

If you know you have boolean arguments, you can get away with using NumPy's bitwise operators, but be careful with parentheses, like this: z = (x > 1) & (x < 2). The absence of NumPy operator forms of logical_and and logical_or is an unfortunate consequence of Python's design.

RESHAPE and LINEAR INDEXING: Matlab always allows multi-dimensional arrays to be accessed using scalar or linear indices, NumPy does not. Linear indices are common in Matlab programs, e.g. find() on a matrix returns them, whereas NumPy's find behaves differently. When converting Matlab code it might be necessary to first reshape a matrix to a linear sequence, perform some indexing operations and then reshape back. As reshape (usually) produces views onto the same storage, it should be possible to do this fairly efficiently. Note that the scan order used by reshape in NumPy defaults to the 'C' order, whereas Matlab uses the Fortran order. If you are simply converting to a linear sequence and back this doesn't matter. But if you are converting reshapes from Matlab code which relies on the scan order, then this Matlab code: z = reshape(x,3,4); should become z = x.reshape(3,4,order='F').copy() in NumPy.

For example you might make a startup script that looks like this (Note: this is just an example, not a statement of "best practices"):

```
# Make all numpy available via shorter 'num' prefix
import numpy as num
# Make all matlib functions accessible at the top level via M.func()
import numpy.matlib as M
# Make some matlib functions accessible directly at the top level via, e.g. rand(3,3)
from numpy.matlib import rand,zeros,ones,empty,eye
# Define a Hermitian function
def hermitian(A, **kwargs):
    return num.transpose(A,**kwargs).conj()
# Make some shortcuts for transpose,hermitian:
#    num.transpose(A) --> T(A)
#    hermitian(A) --> H(A)
T = num.transpose
H = hermitian
```