This is an archival dump of old wiki content --- see scipy.org for current material.
Please see https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Contents

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. NumPyProConPage is another page for curious people who are thinking of adopting Python with NumPy and SciPy instead of MATLAB® and want to see a list of pros and cons.

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 elementwise operations. However, there is a special `matrix` type for doing linear algebra, which is just a subclass of the `array` class. Operations on matrix-class arrays are linear algebra 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.
In MATLAB®, every function must be in a file of the same name, and you can't define local functions in an ordinary script file or at the command-prompt (inlines are not real functions but macros, like in C).	NumPy code is Python code, so it has no such restrictions. You can define functions wherever you like.
MATLAB® has an active community and there is lots of code available for free. But the vitality of the community is limited by MATLAB®'s cost; your MATLAB® programs can be run by only a few.	!NumPy/!SciPy also has an active community, based right here on this web site! It is smaller, but it is growing very quickly. In contrast, Python programs can be redistributed and used freely. See Topical_Software for a listing of free add-on application software, Mailing_Lists for discussions, and the rest of this web site for additional community contributions. We encourage your participation!
MATLAB® has an extensive set of optional, domain-specific add-ons ('toolboxes') available for purchase, such as for signal processing, optimization, control systems, and the whole SimuLink® system for graphically creating dynamical system models.	There's no direct equivalent of this in the free software world currently, in terms of range and depth of the add-ons. However the list in Topical_Software certainly shows a growing trend in that direction.

MATLAB® has a sophisticated 2-d and 3-d plotting system, with user interface widgets.	Addon software can be used with Numpy to make comparable plots to MATLAB®. Matplotlib is a mature 2-d plotting library that emulates the MATLAB® interface. PyQwt allows more robust and faster user interfaces than MATLAB®. And mlab, a "matlab-like" API based on Mayavi2, for 3D plotting of Numpy arrays. See the Topical_Software page for more options, links, and details. There is, however, no definitive, all-in-one, easy-to-use, built-in plotting solution for 2-d and 3-d. This is an area where Numpy/Scipy could use some work.
MATLAB® provides a full development environment with command interaction window, integrated editor, and debugger.	Numpy does not have one standard IDE. However, the IPython environment provides a sophisticated command prompt with full completion, help, and debugging support, and interfaces with the Matplotlib library for plotting and the Emacs/XEmacs editors.
MATLAB® itself costs thousands of dollars if you're not a student. The source code to the main package is not available to ordinary users. You can neither isolate nor fix bugs and performance issues yourself, nor can you directly influence the direction of future development. (If you are really set on Matlab-like syntax, however, there is Octave, another numerical computing environment that allows the use of most Matlab syntax without modification.)	NumPy and SciPy are free (both beer and speech), whoever you are.

'array' or 'matrix'? Which should I use?

Short answer

Use arrays.

- They are the standard vector/matrix/tensor type of numpy. Many numpy function return arrays, not matrices.
- There is a clear distinction between element-wise operations and linear algebra operations.
- You can have standard vectors or row/column vectors if you like.

The only disadvantage of using the array type is that you will have to use `dot` instead of `*` to multiply (reduce) two tensors (scalar product, matrix vector multiplication etc.).

Long answer

Numpy contains both an `array` class and a `matrix` class. The `array` class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while `matrix` is intended to facilitate linear algebra computations specifically. In practice there are only a handful of key differences between the two.

- Operator `*`, `dot()`, and `multiply()`:
  - For `array`, **'`*`' means element-wise multiplication**, and the `dot()` function is used for matrix multiplication.
  - For `matrix`, **'`*`' means matrix multiplication**, and the `multiply()` function is used for element-wise multiplication.
- Handling of vectors (rank-1 arrays)
  - For `array`, the **vector shapes 1xN, Nx1, and N are all different things**. Operations like `A[:,1]` return a rank-1 array of shape N, not a rank-2 of shape Nx1. Transpose on a rank-1 `array` does nothing.
  - For `matrix`, **rank-1 arrays are always upconverted to 1xN or Nx1 matrices** (row or column vectors). `A[:,1]` returns a rank-2 matrix of shape Nx1.
- Handling of higher-rank arrays (rank > 2)
  - `array` objects **can have rank > 2**.
  - `matrix` objects **always have exactly rank 2**.
- Convenience attributes
  - `array` **has a .T attribute**, which returns the transpose of the data.
  - `matrix` **also has .H, .I, and .A attributes**, which return the conjugate transpose, inverse, and `asarray()` of the matrix, respectively.
- Convenience constructor
  - The `array` constructor **takes (nested) Python sequences as initializers**. As in, `array([[1,2,3],[4,5,6]])`.
  - The `matrix` constructor additionally **takes a convenient string initializer**. As in `matrix("[1 2 3; 4 5 6]")`.

There are pros and cons to using both:

- **`array`**
  - 🙂 You can treat rank-1 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`.
    - 🙂 Element-wise multiplication is easy: `A*B`.
    - 🙂 `array` is the "default" NumPy type, so it gets the most testing, and is the type most likely to be returned by 3rd party code that uses NumPy.
    - 🙂 Is quite at home handling data of any rank.
    - 🙂 Closer in semantics to tensor algebra, if you are familiar with that.
    - 🙂 *All* operations (`*`, `/`, `+`, `**` etc.) are elementwise
- **`matrix`**
    - 😕 Behavior is more like that of MATLAB® matrices.
    - 🙁 Maximum of rank-2. To hold rank-3 data you need `array` or perhaps a Python list of `matrix`.
    - 🙁 Minimum of rank-2. You cannot have vectors. They must be cast as single-column or single-row matrices.
    - 🙁 Since `array` is the default in NumPy, some functions may return an `array` even if you give them a `matrix` as an argument. This shouldn't happen with NumPy functions (if it does it's a bug), but 3rd party code based on NumPy may not honor type preservation like NumPy does.
    - 🙂 `A*B` is matrix multiplication, so more convenient for linear algebra.
    - 🙁 Element-wise multiplication requires calling a function, `multipy(A,B)`.
    - 🙁 The use of operator overloading is a bit illogical: `*` does not work elementwise but `/` does.

The `array` is thus much more advisable to use, but in the end, you don't really have to choose one or the other. You can mix-and-match. You can use `array` for the bulk of your code, and switch over to `matrix` in the sections where you have nitty-gritty linear algebra with lots of matrix-matrix multiplications.

Facilities for Matrix Users

Numpy has some features that facilitate the use of the `matrix` type, which hopefully make things easier for Matlab converts.

- A `matlib` module has been added that contains matrix versions of common array constructors like `ones()`, `zeros()`, `empty()`, `eye()`, `rand()`, `repmat()`, etc. Normally these functions return `array`s, but the `matlib` versions return `matrix` objects.
- `mat` has been changed to be a synonym for `asmatrix`, rather than `matrix`, thus making it concise way to convert an `array` to a `matrix` without copying the data.
- Some top-level functions have been removed. For example `numpy.rand()` now needs to be accessed as `numpy.random.rand()`. Or use the `rand()` from the `matlib` module. But the "numpythonic" way is to use `numpy.random.random()`, which takes a tuple for the shape, like other numpy functions.

Table of Rough MATLAB-NumPy Equivalents

The table below gives rough equivalents for some common MATLAB® expressions. **These are not exact equivalents**, but rather should be taken as hints to get you going in the right direction. For more detail read the built-in documentation on the NumPy functions.

Some care is necessary when writing functions that take arrays or matrices as arguments --- if you are expecting an `array` and are given a `matrix`, or vice versa, then '*' (multiplication) will give you unexpected results. You can convert back and forth between arrays and matrices using

- **`asarray`**: always returns an object of type `array`
- **`asmatrix`** or **`mat`**: always return an object of type `matrix`
- **`asanyarray`**: always returns an `array` object or a subclass derived from it, depending on the input. For instance if you pass in a `matrix` it returns a `matrix`.

These functions all accept both arrays and matrices (among other things like Python lists), and thus are useful when writing functions that should accept any array-like object.

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

Toggle line numbers

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

Also assume below that if the Notes talk about "matrix" that the arguments are rank 2 entities.

THIS IS AN EVOLVING WIKI DOCUMENT. If you find an error, or can fill in an empty box, please fix it! If there's something you'd like to see added, just add it.

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`	`spacing(1)`	Distance between 1 and the nearest floating point number
`ode45`	`scipy.integrate.ode(f).set_integrator('dopri5')`	integrate an ODE with Runge-Kutta 4,5
`ode15s`	`scipy.integrate.ode(f).\ set_integrator('vode', method='bdf', order=15)`	integrate an ODE with BDF

Linear Algebra Equivalents

The notation `mat(...)` means to use the same expression as array, but convert to matrix with the `mat()` type converter.

The notation `asarray(...)` means to use the same expression as matrix, but convert to array with the `asarray()` type converter.

MATLAB	numpy.array	numpy.matrix	Notes
`ndims(a)`	`ndim(a)` or `a.ndim`		get the number of dimensions of a (tensor rank)
`numel(a)`	`size(a)` or `a.size`		get the number of elements of an array
`size(a)`	`shape(a)` or `a.shape`		get the "size" of the matrix
`size(a,n)`	`a.shape[n-1]`		get the number of elements of the *n*th dimension of array a. (Note that MATLAB® uses 1 based indexing while Python uses 0 based indexing, *See note 'INDEXING'*)
`[ 1 2 3; 4 5 6 ]`	`array([[1.,2.,3.], [4.,5.,6.]])`	`mat([[1.,2.,3.], [4.,5.,6.]])` or `mat("1 2 3; 4 5 6")`	2x3 matrix literal
`[ a b; c d ]`	`vstack([hstack([a,b]), hstack([c,d])])`	`bmat('a b; c d')`	construct a matrix from blocks a,b,c, and d
`a(end)`	`a[-1]`	`a[:,-1][0,0]`	access last element in the 1xn matrix `a`
`a(2,5)`	`a[1,4]`		access element in second row, fifth column
`a(2,:)`	`a[1]` or `a[1,:]`		entire second row of `a`
`a(1:5,:)`	`a[0:5]` or `a[:5]` or `a[0:5,:]`		the first five rows of `a`
`a(end-4:end,:)`	`a[-5:]`		the last five rows of `a`
`a(1:3,5:9)`	`a[0:3][:,4:9]`		rows one to three and columns five to nine of `a`. This gives read-only access.

`a([2,4,5],[1,3])`	`a[ix_([1,3,4],[0,2])]`		rows 2,4 and 5 and columns 1 and 3. This allows the matrix to be modified, and doesn't require a regular slice.
`a(3:2:21,:)`	`a[ 2:21:2,:]`		every other row of `a`, starting with the third and going to the twenty-first
`a(1:2:end,:)`	`a[ ::2,:]`		every other row of `a`, starting with the first
`a(end:-1:1,:)` or `flipud(a)`	`a[ ::-1,:]`		`a` with rows in reverse order
`a([1:end 1],:)`	`a[r_[:len(a),0]]`		`a` with copy of the first row appended to the end
`a.'`	`a.transpose()` or `a.T`		transpose of `a`
`a'`	`a.conj().transpose()` or `a.conj().T`	`a.H`	conjugate transpose of `a`
`a * b`	`dot(a,b)`	`a * b`	matrix multiply
`a .* b`	`a * b`	`multiply(a,b)`	element-wise multiply
`a./b`	`a/b`		element-wise divide
`a.^3`	`a**3`	`power(a,3)`	element-wise exponentiation
`(a>0.5)`	`(a>0.5)`		matrix whose i,jth element is (a_ij > 0.5)
`find(a>0.5)`	`nonzero(a>0.5)`		find the indices where (a > 0.5)
`a(:,find(v>0.5))`	`a[:,nonzero(v>0.5)[0]]`	`a[:,nonzero(v.A>0.5)[0]]`	extract the columms of a where vector v > 0.5
`a(:,find(v>0.5))`	`a[:,v.T>0.5]`	`a[:,v.T>0.5)]`	extract the columms of a where column vector v > 0.5
`a(a<0.5)=0`	`a[a<0.5]=0`		a with elements less than 0.5 zeroed out
`a .* (a>0.5)`	`a * (a>0.5)`	`mat(a.A * (a>0.5).A)`	a with elements less than 0.5 zeroed out
`a(:) = 3`	`a[:] = 3`		set all values to the same scalar value
`y=x`	`y = x.copy()`		numpy assigns by reference
`y=x(2,:)`	`y = x[1,:].copy()`		numpy slices are by reference
`y=x(:)`	`y = x.flatten(1)`		turn array into vector (note that this forces a copy)
`1:10`	`arange(1.,11.)` or `r_[1.:11.]` or `r_[1:10:10j]`	`mat(arange(1.,11.))` or `r_[1.:11.,'r']`	create an increasing vector *see note 'RANGES'*
`0:9`	`arange(10.)` or `r_[:10.]` or `r_[:9:10j]`	`mat(arange(10.))` or `r_[:10.,'r']`	create an increasing vector *see note 'RANGES'*
`[1:10]'`	`arange(1.,11.)[:, newaxis]`	`r_[1.:11.,'c']`	create a column vector
`zeros(3,4)`	`zeros((3,4))`	`mat(...)`	3x4 rank-2 array full of 64-bit floating point zeros

zeros(3,4,5)	zeros((3,4,5))	mat(...)	3x4x5 rank-3 array full of 64-bit floating point zeros
ones(3,4)	ones((3,4))	mat(...)	3x4 rank-2 array full of 64-bit floating point ones
eye(3)	eye(3)	mat(...)	3x3 identity matrix
diag(a)	diag(a)	mat(...)	vector of diagonal elements of a
diag(a,0)	diag(a,0)	mat(...)	square diagonal matrix whose nonzero values are the elements of a
rand(3,4)	random.rand(3,4)	mat(...)	random 3x4 matrix
linspace(1,3,4)	linspace(1,3,4)	mat(...)	4 equally spaced samples between 1 and 3, inclusive
[x,y]=meshgrid(0:8,0:5)	mgrid[0:9.,0:6.] or meshgrid(r_[0:9.],r_[0:6.]	mat(...)	two 2D arrays: one of x values, the other of y values
	ogrid[0:9.,0:6.] or ix_(r_[0:9.],r_[0:6.]	mat(...)	the best way to eval functions on a grid
[x,y]=meshgrid([1,2,4],[2,4,5])	meshgrid([1,2,4],[2,4,5])	mat(...)	
	ix_([1,2,4],[2,4,5])	mat(...)	the best way to eval functions on a grid
repmat(a, m, n)	tile(a, (m, n))	mat(...)	create m by n copies of a
[a b]	concatenate((a,b),1) or hstack((a,b)) or column_stack((a,b)) or c_[a,b]	concatenate((a,b),1)	concatenate columns of a and b
[a; b]	concatenate((a,b)) or vstack((a,b)) or r_[a,b]	concatenate((a,b))	concatenate rows of a and b
max(max(a))	a.max()		maximum element of a (with ndims(a) <=2 for matlab)
max(a)	a.max(0)		maximum element of each column of matrix a
max(a,[],2)	a.max(1)		maximum element of each row of matrix a
max(a,b)	maximum(a, b)		compares a and b element-wise, and returns the maximum value from each pair
norm(v)	sqrt(dot(v,v)) or Sci.linalg.norm(v) or linalg.norm(v)	sqrt(dot(v.A,v.A)) or Sci.linalg.norm(v) or linalg.norm(v)	L2 norm of vector v
a & b	logical_and(a,b)		element-by-element AND operator (Numpy ufunc) *see note 'LOGICOPS'*
a | b	logical_or(a,b)		element-by-element OR operator (Numpy ufunc) *see note 'LOGICOPS'*
bitand(a,b)	a & b		bitwise AND operator (Python native and Numpy ufunc)
bitor(a,b)	a | b		bitwise OR operator

			(Python native and Numpy ufunc)
`inv(a)`	`linalg.inv(a)`		inverse of square matrix a
`pinv(a)`	`linalg.pinv(a)`		pseudo-inverse of matrix a
`rank(a)`	`linalg.matrix_rank(a)`		rank of a matrix a
`a\b`	`linalg.solve(a,b)` if a is square `linalg.lstsq(a,b)` otherwise		solution of a x = b for x
`b/a`	Solve a.T x.T = b.T instead		solution of x a = b for x
`[U,S,V]=svd(a)`	`U, S, Vh = linalg.svd(a), V = Vh.T`		singular value decomposition of a
`chol(a)`	`linalg.cholesky(a).T`		cholesky factorization of a matrix (chol(a) in matlab returns an upper triangular matrix, but linalg.cholesky(a) returns a lower triangular matrix)
`[V,D]=eig(a)`	`D,V = linalg.eig(a)`		eigenvalues and eigenvectors of a
`[V,D]=eig(a,b)`	`V,D = Sci.linalg.eig(a,b)`		eigenvalues and eigenvectors of a,b
`[V,D]=eigs(a,k)`			find the k largest eigenvalues and eigenvectors of a
`[Q,R,P]=qr(a,0)`	`Q,R = Sci.linalg.qr(a)`	`mat(...)`	QR decomposition
`[L,U,P]=lu(a)`	`L,U = Sci.linalg.lu(a)` or `LU,P=Sci.linalg.lu_factor(a)`	`mat(...)`	LU decomposition (note: P(Matlab) == transpose(P(numpy)))
`conjgrad`	`Sci.linalg.cg`	`mat(...)`	Conjugate gradients solver
`fft(a)`	`fft(a)`	`mat(...)`	Fourier transform of a
`ifft(a)`	`ifft(a)`	`mat(...)`	inverse Fourier transform of a
`sort(a)`	`sort(a)` or `a.sort()`	`mat(...)`	sort the matrix
`[b,I] = sortrows(a,i)`	`I = argsort(a[:,i]), b=a[I,:]`		sort the rows of the matrix
`regress(y,X)`	`linalg.lstsq(X,y)`		multilinear regression
`decimate(x, q)`	`Sci.signal.resample(x, len(x)/q)`		downsample with low-pass filtering
`unique(a)`	`unique(a)`		
`squeeze(a)`	`a.squeeze()`		

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. Python also has an `inspect` module (do `import inspect`) which provides a `getfile` that often works.

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. Confusion and flamewars arise because each has advantages and disadvantages. One based indexing is consistent with common human language usage, where the "first" element of a sequence has index 1. Zero based indexing simplifies indexing. See also a text by prof.dr. Edsger W. Dijkstra.

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.

Customizing Your Environment

In MATLAB® the main tool available to you for customizing the environment is to modify the search path with the locations of your favorite functions. You can put such customizations into a startup script that MATLAB will run on startup.

NumPy, or rather Python, has similar facilities.

- To modify your Python search path to include the locations of your own modules, define the `PYTHONPATH` environment variable.
- To have a particular script file executed when the interactive Python interpreter is started, define the `PYTHONSTARTUP` environment variable to contain the name of your startup script.

Unlike MATLAB®, where anything on your path can be called immediately, with Python you need to first do an 'import' statement to make functions in a particular file accessible.

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

Toggle line numbers

```python
# 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 shorcuts for transpose,hermitian:
#    num.transpose(A) --> T(A)
#    hermitian(A) --> H(A)
T = num.transpose
H = hermitian
```

MATLAB packages/tools and equivalent for use with NumPy

- **Plotting**: matplotlib provides a workalike interface for 2D plotting; Mayavi provides 3D plotting
- **Symbolic calculation**: swiginac appears to be the most complete current option. sympy is a project aiming at bringing the basic symbolic calculus functionalities to Python. Also to be noted is PyDSTool which provides some basic symbolic functionality.
- **Linear algebra**: scipy.linalg provides the LAPACK routines
- **Interpolation**: [/ScipyPackages/Interpolate scipy.interpolate] provides several spline interpolation tools
- **Numerical integration**: scipy.integrate provides several tools for integrating functions as well as some basic ODE integrators. Convert XML vector field specifications automatically using VFGEN.

- **Dynamical systems**: PyDSTool provides a large dynamical systems and modeling package, including good ODE/DAE integrators. Convert XML vector field specifications automatically using VFGEN.
- **Simulink**: no alternative is currently available.

Links

See http://mathesaurus.sf.net/ for another MATLAB®/NumPy cross-reference.

See http://urapiv.wordpress.com for an open-source project (URAPIV) that attempts to move from MATLAB® to Python (PyPIV http://sourceforge.net/projects/pypiv) with SciPy / NumPy.

In order to create a programming environment similar to the one presented by MATLAB®, the following are useful:

- IPython: an interactive environment with many features geared towards efficient work in typical scientific usage very similar (with some enhancments) to MATLAB® console.
- Matplotlib: a 2D ploting package with a list of commands similar to the ones found in matlab. Matplotlib is very well integrated with IPython.
- Spyder a free and open-source Python development environment providing a MATLAB®-like interface and experience
- SPE is a good free IDE for python. Has an interactive prompt.
- Eclipse: is one nice option for python code editing via the pydev plugin.
- Wing IDE: a commercial IDE available for multiple platforms. The professional version has an interactive debugging prompt similar to MATLAB's.
- Python(x,y) scientific and engineering development software for numerical computations, data analysis and data visualization. The installation includes, among others, Spyder, Eclipse and a lot of relevant Python modules for scientific computing.
- Python Tools for Visual Studio: a rich IDE plugin for Visual Studio that supports CPython, IronPython, the IPython REPL, Debugging, Profiling, including running debugging MPI program on HPC clusters.

An extensive list of tools for scientific work with python is in the link: Topical_Software.

MATLAB® and SimuLink® are registered trademarks of The MathWorks.

CategoryCookbook CategoryTemplate CategoryTemplate CategoryCookbook CategorySciPyPackages CategoryCategory