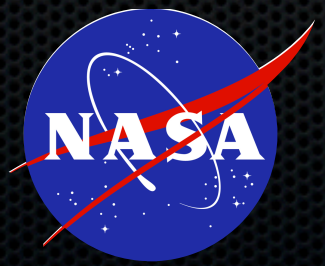
Scientific Computing



Daniel Kocevski

NASA Goddard Space Flight Center

Lecture Download

- http://www.kocevski.com/Temp/SciPy.pdf

Numpy & Matplotlib Review

- NumPy
  - Fundamental package for scientific computing with Python
  - A powerful N-dimensional array object
  - Sophisticated (broadcasting) functions
  - Comparison testing, selection, and manipulation
  - Linear algebra, Fourier transform, and random number capabilities
  - Basic statistics
- Matplotlib
  - Basic 2D (MATLAB like) plotting capabilities

SciPy Overview

- Collection of algorithms and mathematical tools
  - Numpy & matplot lib are subsets of Scipy
- Contains modules for
  - Linear algebra, integration, & interpolation
  - FFT, signal and image processing
  - ODE solvers, special functions, and optimization
- The power of IDL, MATLAB, etc comes from built-in analysis libraries
  - SciPy gives python those capabilities

Core Packages

- Core SciPy Library
  - Numerical recipes-like library
- Numpy
  - Base N-dimensional arrays
- Matplotlib
  - Comprehensive 2D plotting
- iPython
  - Enhanced interactive console
- Sympy
  - Symbolic mathematics
- Pandas
  - Data structures and analysis

Core Library Subpackages

- Constants: physical constants and conversion factors
- Cluster: hierarchical clustering, vector quantization, K-means
- fftpack: Discrete Fourier Transform algorithms
- integrate: numerical integration routines
- interpolate: interpolation tools
- io: data input and output
- lib: Python wrappers to external libraries
- linalg: linear algebra routines
- misc: miscellaneous utilities (e.g. image reading/writing)
- ndimage: various functions for multi-dimensional image processing
- optimize: optimization algorithms including linear programming
- signal: signal processing tools
- sparse: sparse matrix and related algorithms
- spatial: KD-trees, nearest neighbors, distance functions
- special: special functions
- stats: statistical functions
- weave: tool for writing C/C++ code as Python multiline strings
- RPy: interface to the R statistical package for advanced data analysis.

Building N-dimensional arrays

```
>>> import numpy as np
>>> import scipy as sp

>>> sp.mgrid[0:5]
array([0, 1, 2, 3, 4])
>>> sp.mgrid[0:1:5j]
array([ 0., 0.25, 0.5, 0.75, 1.])

>>> sp.mgrid[0:3,0:3]
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],
       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])

>>> np.mgrid[0:3,0:3].shape
(2, 3, 3)
```

- We talked about methods to fill arrays
  - arange, ones, zeros, linspace
- SciPy has additional methods
  - mgrid[ ]
    - Form and fill higher dimensional arrays

Array Concatenation

- What if you want to combine two arrays?
- Use sp.r_
- Combined 1d or Nd arrays

```
>>> import scipy as sp

>>> x = sp.arange(3)
array([0, 1, 2])
>>> y = sp.ones(3)
array([1, 1, 1])
>>> z = sp.r_(x,y)
array([0, 1, 2, 1, 1, 1])
>>> x = np.array([[1,2,3],[4,5,6]
array([[1, 2, 3],
       [4, 5, 6]])
>>> sp.r_(x,x)
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
>>> sp.r_('-1',x,x)
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Vectorizing functions

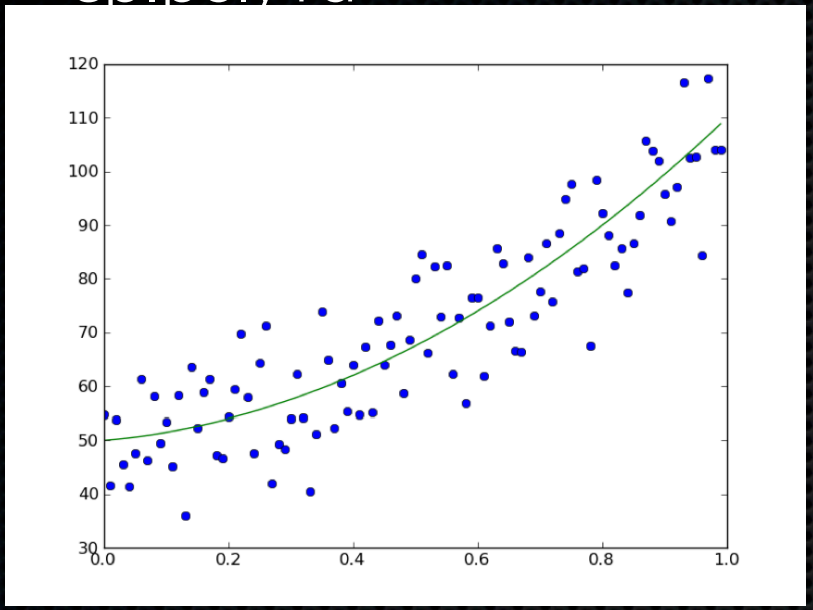
```
>>> def myfun(a,b):
...     if (a<b):
...       return a
...     else:
...       return 1.0

>>> x = np.arange(5)/10.
>>> x
array([ 0., 0.1, 0.2, 0.3, 0.4])
>>> vec_myfun = sp.vectorize(myfun)
>>> vec_myfun(x,0.3)
array([ 0. , 0.1, 0.2, 1., 1. ])
```

- Methods to vectorize functions
  - sp.vectorize
- Takes a nested sequence of objects or numpy arrays as inputs and returns a numpy array as output.

Generating Fake Data

- sp.poly1d




```
>>> import matplotlib.pylab as plt

>>> time = np.arange(100)/100.
>>> par0 = [50.,10,50.]
>>> poly = np.poly1d(par0)
>>> print poly
50 x + 10 x + 50
>>> type(poly)
numpy.lib.polynomial.poly1d

>>> model = sp.polyval(par0,time)
>>> err = np.sqrt(model)
>>> data = model + (err *
np.random.normal(size=len(err)))
>>> plt.plot(time,data,'o',time,model)
>>> plt.show()
```

Statistics

```
>>> data.mean() # or mean(data)
70.602942080945425
>>> np.median(data) 67.598541045505613
>>> np.std(data)
18.784744925971221

>>> data = np.random.normal(0,1,size=(5))
>>> median(data,axis=0)
array([ 0.82514088, -0.3784086 ,
-0.05834713, 0.01386631, -0.22942265 ])
```

- All your favorites are available
- Running median filter exists as scipy.signal.medfilt

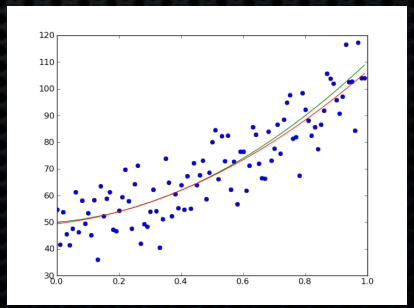
Data Fitting

- Least squares polynomial fit
  - sp.polyfit
- Several data sets of sample points can be fit at once by passing in a 2D-array
- You can assign weights, but errors are not considered

```
>>> import matplotlib.pylab as plt

>>> time = np.arange(100)/100.
>>> par0 = [50.,10,50.]
>>> model = sp.polyval(par0,time)
>>> err = np.sqrt(data)
>>> data = model + (err *
np.random.normal(size=len(err)))

>>> fit = np.polyfit(time,data,2)
>>> plt.plot(time,data,'o',time,model)
>>> plt.plot(time,np.polyval(fit,time))
>>> plt.show()
```



Data Fitting with Errors

```
>>> from scipy.optimize import
fmin, leastsq

>>> def resid_fun(param):
...     return (data-
np.polyval(param,time))/err

>>> def chi_fun(param):
...     return
(resid_fun(param)**2).sum()

>>> par_fmin = fmin(chi_fun,par)
>>> par_lstq =
leastsq(resid_fun,par,full_output=
True)
```

- scipy.optimize
  - Allows for more advanced data fitting
- fmin: Minimize a function using the downhill simplex algorithm
- leastsq: Minimize the sum of squares of a set of equations
- Much more!

Parameter Uncertainties

```
>>> from scipy.stats import norm

>>> par_lstq[0]
array([ 55.18927249, 5.39642436, 51.46318436])
>>> dpar_lstq = np.sqrt( np.diag( par_lstq[1] ) )
array([ 11.27959567, 10.95999741, 2.17067466])

>>> diff = (par_lstq[0]-par0) / dpar_lstq
>>> diff
array([ 0.46005838, -0.42003437, 0.67406894])
>>> pval = 2. - 2.*norm.cdf( np.abs(diffs) )
array([ 0.64547432, 0.67446034, 0.50026749])
```

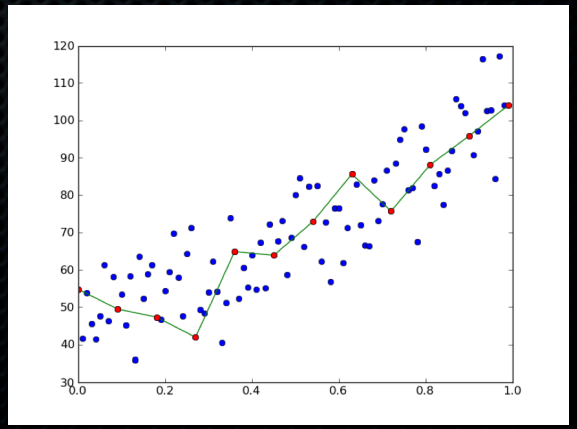
- Can get parameter uncertainties from scipy.leastsq
- Could get p-values for these from scipy.stats

Interpolation

```
>>> import matplotlib.pylab as plt
>>> from scipy.interpolate import
interp1d

>>> time_skip = time[::9]
>>> data_skip = data[::9]
>>> res = interp1d(time_skip,data_skip)
>>> plt.plot(time,data,'o')
>>> plt.plot(time,res(time))
>>> plt.plot(time_skip,data_skip,'o')
```

- Interpolate a 1-D function
  - interp1d
- Fancier interpolation schemes are available in scipy.signal



Special Functions

- All of your favorite special functions for any occasion!

```
>>> from scipy import special
>>> special?
Airy Functions
Elliptic Functions and Integrals
Bessel Functions
Zeros of Bessel Functions
Integrals of Bessel Functions
Derivatives of Bessel Functions
Spherical Bessel Functions
Ricatti-Bessel Functions
Struve Functions
Raw Statistical Functions (Friendly
versions in scipy.stats) Gamma and Related
Functions
Error Function and Fresnel Integrals
Legendre Functions
Orthogonal polynomials --- 15 types
HyperGeometric Functions
```

Physical Constants

```
>>> from scipy import constants
>>> dir(constants)
['Avogadro',
'Bolzmann', 'Btu', 'Btu_IT', 'Btu_th',
'C2F',
...
...
'yotta',
'zebi',
'zepto', 'zero_Celsius', 'zetta']

>>> constants.Avogadro
6.022141500000003e+23
>>> constants.C2F(0.)
32.0
```

- All of your old friends are there waiting for you to say hello!

Numerical Integration

- Integrating Functions
  - quad: Compute a definite integral
  - fixed_quad: Compute a definite integral using fixed-order Gaussian quadrature
- Fixed Sample Integrals
  - cumtrapz: Trapezoidal rule
  - simps: Simpson's rule

```
>>> from scipy.integrate import
quad
>>> quad( lambda x: sin(x), 0,
np.pi) (2.0,
2.220446049250313e-14)

>>> x = np.arange(10)
>>> y = x**2
>>> int = simps(y,x)
243.16666666666663
```

Inline C Code

```
>>> from scipy import weave

>>> x = np.arange(1000)/1000.
>>> y = np.empty(1000,dtype=np.double)
>>> n=len(x)
>>> code = """
... int i;
...   for (i=0;i<n;i++) {
...     if (*x<0.5) *y = exp(-(*x)*2);
...     else *y = exp(-(*x));
...     x++; y++;
...   } """

>>> weave.inline(code,['n','x','y'])
```

- You can run C code in python through the weave module

Weave Speed Comparison

```
>>> def moronic_function(x,y):
...  lx=len(x)
...  for k in xrange(lx):
...    if (x[k]<0.5):
...      y[k]=np.exp(-x[k]*2)
...    else:
...      y[k]=np.exp(-x[k])
...

>>> timeit moronic_function(x,y)
100 loops, best of 3: 4.32 ms per loop
>>> timeit weave.inline(code,['n','x','y'])
10000 loops, best of 3: 36.8 us per loop
```

- The C code is much faster than the Python code!

Further Reading

- SciPy Reference
  - http://docs.scipy.org/doc/scipy/scipy-ref.pdf
- Numpy for IDL users:
  - https://www.cfa.harvard.edu/~jbattat/computer/python/science/idl-numpy.html
- Numpy for MATLAB users:
  - http://www.scipy.org/NumPy_for_Matlab_Users

Breakout Session

- Generate fake data using a period function
- Add scatter to the fake data
- Interpolate the data on a timescale < the period
- Calculate the period using a fast fourier analysis
- Perform a non-linear model fit to a subset of the fake data