Coding Bootcamp Code in Python

PYTHON FOR SCIENTIFIC COMPUTING: NUMPY

Out of the box

- Python is interpreted
  - Python is slow
  - Python is really slow
- Okay for one-offs, prototypes, short runtimes
- ***Not okay*** for computationally intensive tasks!

Don't use vanilla Python for computations!!!

Python performance

500 × 500 matrices

Python: 0.09 s

C: 0.014 s

Fortran: 0.012 s

```
def init_matrix(n):
    m = []
    for i in range(n):
        m.append([])
        for j in range(n):
            m[i].append(random.random())
    return m

def matmul(a, b, c):
    n = len(a)
    for i in range(n):
        for j in range(n):
            c[i][j] = 0.0
            for k in range(n):
                c[i][j] += a[i][k]*b[k][j]
```

Represent matrix as list of lists

Naive!

$$C = A \cdot B$$

$$C_{ij} = A_{i1}B_{1j} + \cdots + A_{iN}B_{Nj}$$

Python: 32 s

C: 0.49 s

Fortran: 0.11 s

Libraries for numeric computation

- numpy
  - Fast arrays
  - Matrix operations (BLAS-like)
  - Linear algebra
  - Fast Fourier Transform
  - Mathematical functions defined on arrays
  - Pseudo-random number generation to initialize arrays
  - Simple statistics
- …

Python using numpy

500 × 500 matrices

numpy: 0.011 s

numpy: 0.077 s

```
import numpy as np

def init_matrix(n):
    return np.random.uniform(0.0, 1.0, (n, n))

def matmul(a, b):
    return a @ b
```

Language/library	Execution time matrix multiplication (s)
Python	32
C	0.49
Fortran	0.11
Python/numpy	0.077
Fortran/BLAS	0.060

415 ×

Creating array I

convention

```
import numpy as np
```

```
v1 = np.zeros(3)
v2 = np.ones(3)
v3 = np.empty(3)
```

create 3-element array, all 0.0

create 3-element array, all 1.0

create 3-element "empty" array
initialize elements later

```
a = np.zeros((2, 3))
b = np.ones((2, 3))
c = np.eye(2)
d = np.empty((2, 3))
```

create 2 × 3 array, all elements 0.0

create 2 × 3 array, all elements 1.0

create 2 × 2 identity array

create 2 × 3 "empty" array
initialize elements later

Default type: np.float ≡ double precision

Creating arrays II

```
e = np.random.uniform(0.0, 1.0, (2, 3))
```

create 2×3 array, elements x randomly drawn from uniform distribution such that $x \in [0.0, 1.0[$

```
f = np.array([[3.1, 4.2, -1.1], [-0.3, 1.3, 13.1]])
```

create 2×3 array from a Python list of lists

```
f = np.genfromtxt('matrix.txt')
```

create 2×3 array from text file

```
1.2 2.3 3.4
4.5 5.6 6.7
```

Creating arrays III

```
e = np.arange(-1.0, 1.0, 0.25)
```

create 8-element array, first element -1.0,
last element less than 1.0, step 0.25

```
[ -1.   -0.75  -0.5  -0.25  0.   0.25  0.5  0.75 ]
```

```
f = np.linspace(-1.0, 1.0, 9)
```

create 9-element array, first element -1.0,
last element 1.0, determine step

```
[ -1.   -0.75  -0.5  -0.25  0.   0.25  0.5  0.75  1. ]
```

Numpy data types

- Integers
  `np.int8`, `np.int16`, `np.int32`, `np.int64`
  default for `np.int`: `np.int32` on 32-bit, `np.int64` on 64-bit architecture (`np.uint`<*n*> for unsigned integers)
- Floating point numbers
  `np.float16, p.float32`, `np.float64`,
  `np.float96`
  default for `np.float`: `np.float64`, i.e., double precision
- Complex numbers
  `np.complex64`, `np.complex128`, `np.complex192`
  default for `np.complex`: `np.complex128`, i.e., double precision
- Boolean values: `np.bool`
- Characters/

```
v = np.zeros(3, dtype=np.int8)
```

Accessing array elements

```
a = np.zeros((2, 3))
```

- Array dimensions, strides

```
a.shape == (2, 3)
```

nr. bytes to next row

```
a.strides == (24, 8)
```

nr. bytes to next column

- Assigning to a specific element

```
a[1, 0] = 5.0
```

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ \boxed{5.0} & 0.0 & 0.0 \end{pmatrix}$$

- Using an element's value

```
q = a[1, 0] + a[1, 2]
```

Note:

- Implicit conversion of tuple for indexing
- 0-based indexing

Accessing subarrays: slicing

```
a = np.arange(1, 21).reshape(4, 5)
```

```
[[  1   2   3   4   5 ]
 [  6   7   8   9  10 ]
 [ 11  12  13  14  15 ]
 [ 16  17  18  19  20 ]]
```

- Second column

```
a[:, 1]
```

```
[ 2  7  12  17 ]
```

- Third row

result: 1D array

```
a[2, :]
```

```
[ 11  12  13  14  15 ]
```

- 2D subarray

```
a[1:3, 1:3] = np.eye(2)
```

```
[[  1   2   3   4   5 ]
 [  6   1   0   9  10 ]
 [ 11   0   1  14  15 ]
 [ 16  17  18  19  20 ]]
```

cfr. list slicing, but…
array slicing does ***not*** copy!

Fancy indexing

```
a = np.arange(1, 10, dtype=np.int).reshape(3, 3)
```

```
[[  1   2   3 ]
 [  4   5   6 ]
 [  7   8   9 ]]
```

- Conditional indexing

```
a[a % 2 == 1] = 0
```

3 × 3 Boolean array

```
[[  0   2   0 ]
 [  4   0   6 ]
 [  0   8   0 ]]
```

- Conditional assignment

```
np.where(a > 0, 1, -1)
```

3 × 3 Boolean array

```
[[ -1   1  -1 ]
 [  1  -1   1 ]
 [ -1   1  -1 ]]
```

Operations on arrays

- Scalar-array operations: +, −, *, /, //, **

```
a = np.array([[1.0, 3.0], [4.0, 5.0]])
print(3.0 + a)
```

```
[[ 4.   6. ]
 [ 7.   8. ]]
```

- Element-wise operations: +, −, *, /, //, **

```
a = np.array([[1.0, 3.0], [4.0, 5.0]])
b = np.array([[2.0, 3.0], [1.0, 0.5]])
print(a*b)
```

```
[[ 2.   9.  ]
 [ 4.   2.5 ]]
```

- Matrix product

```
print(np.dot(a, b))
```

```
[[  5.    4.5 ]
 [ 13.   14.5 ]]
```

  – Python 3.5 style

```
print(a @ b)
```

Functions operating on arrays

```
a = np.empty((500, 500))
b = np.random.uniform(0.0, 1.0, (500, 500))
```

- Avoid

```
for i in range(500):
    for j in range(500):
        a[i, j] = math.sqrt(b[i, j])
```

140 μs

21 ×

6.5 μs

- Use

```
a = np.sqrt(b)
```

- Other functions: `np.sin`, …, `np.sinh`, …, `np.exp`, `np.trace`, `np.transpose`,…

Some linear algebra

```
a = np.array([[1.0, 3.0], [4.0, 5.0]])
```

- numpy has some linear algebra operations
  - matrix power

```
np.linalg.matrix_power(a, 3)
```

```
[[  85.   129.]
 [ 172.   257.]]
```

  - matrix inverse

```
np.linalg.inv(a)
```

```
[[-0.71428571  0.42857143]
 [ 0.57142857 -0.14285714]]
```

  - determinant

```
np.linalg.det(a)
```

```
-7.0
```

  - eigen values

```
np.linalg.eigvals(a)
```

```
[-1.  7.]
```

References versus copies

- Reshape: different view on same data

```
a = np.array([[1.0, 3.0],
              [4.0, 5.0]])
```

```
[[ 1.   3.]
 [ 4.   5.]]
```

```
b = a.reshape((a.size, ))
```

```
[ 1. 3. 4. 5.]
```

```
a[0, 0] = 13.0
```

a

```
[[ 13.   3.]
 [  4.   5.]]
```

b

```
[ 13. 3. 4. 5.]
```

- Some operations return copies, check documentation carefully

numpy data I/O revisited

- Reading text file with 10^9 64-bit floats
  - `np.fromtxt(…)`: 57 minutes, 44 GB RAM
  - `np.fromfile(…, sep='\n')`: 4.6 minutes, 8 GB RAM

5 ×

12 ×

35 ×

All functions are equal, but…
some are more equal than others

- Reading binary file with 10^9 64-bit floats
  - `np.fromfile(…)`: 8 seconds, 8 GB RAM

Not all data formats are equal: HDF5 to the rescue

Matrices

- Matlab-like initialization

```
a = np.matrix('1.0  3.0; 4.0  5.0')
```

```
[[ 1.   3. ]
 [ 4.   5. ]]
```

- Overloaded * and ** operators

```
a = np.matrix([[1.0, 3.0], [4.0, 5.0]])
b = np.matrix([[2.0, 3.0], [1.0, 0.5]])
print(a*b)
print(a**3)
```

```
[[  5.    4.5 ]
 [ 13.   14.5 ]]
```

```
[[  85.   129. ]
 [ 172.   257. ]]
```

Don't use, confusing

- Result is always matrix (2D)

```
a = np.matrix('1.0, 3.0')
b = np.matrix('2.0; 4.0')
print(a*b)
```

```
[[ 14. ]]
```

References

- numpy for MATLAB users
  http://mathesaurus.sourceforge.net/matlab-numpy.html

Code Pack 25

A. Numpy: Make the code

Coding Bootcamp Code in Python

PYTHON FOR SCIENTIFIC COMPUTING: SCIPY

Libraries for numeric computation

- …
- scipy
  - Dense/sparse linear algebra
  - Solving ordinary differential equations
  - Numerical integration
  - Optimization
  - Interpolation
  - Signal processing
  - Statistics
  - Special mathematical functions
  - Mathematical & physical constants
- …

```
import scipy as sp
```

convention

```
import scipy.linalg
```

import subpackages as needed

Singular Value Decomposition

- Computing SVD

Should be fast when built against good BLAS/Lapack library

```
import scipy.linalg
a = np.array([[7.3, 5.7], [-1.2, 5.3]])
u, s, v = sp.linalg.svd(a)
```

- Note: `s` is not a 2D-array, it is a 1D-array

```
S = np.diag(s)
```

- Let's check

```
A = u @ S @ v
delta = A - a
```

```
[[ 8.88178420e-16, 0.00000000e+00],
 [ 4.44089210e-16, 0.00000000e+00]]
```

Linear regression

- Reading data

```
x,y
0.000e+00,1.206e+00
5.263e-02,1.207e+00
…
```

data.csv

```
data = np.genfromtxt('data.csv',
                     dtype=[np.float64, np.float64],
                     delimiter=',', names=True)
```

- Linear regression

```
import scipy as sp
import scipy.stats
slp, intc, r, _, _ = sp.stats.linregress(data['x'],
                                          data['y'])
```

Optimization: function definitions

- Minimize $f(x, y) = (x^2 + y^2)^2 - 2x^2 - 2y^2 + 0.1x$
- Define function

```
def f(X):
    x = X[0]
    y = X[1]
    return (x**2 + y**2)**2 - 2*x**2 - 2*y**2 + 0.1*x
```

- Define gradient

```
def grad_f(X):
    x = X[0]
    y = X[1]
    f_x = 4*(x**2 + y**2)*x - 4*x + 0.1
    f_y = 4*(x**2 + y**2)*y - 4*y
    return np.array([f_x, f_y])
```

Optimization

- Compute minimum

```
import scipy.optimize

x0 = np.array([1.0, 0.01])
xopt = scipy.optimize.fmin_cg(f, x0, fprime=grad_f,
                              disp=False)
```

- Many methods
  - Powell
  - Conjugate gradient
  - BFGS
  - Newton conjugate gradient
  - …

Ordinary differential equations

- Rewrite higher order differential equation to set of first order equations

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\theta - q\omega + F_D \sin\Omega_D t \quad \Leftrightarrow \begin{cases} \dfrac{d\theta}{dt} = \omega \\ \dfrac{d\omega}{dt} = -\dfrac{g}{l}\theta - q\omega + F_D \sin\Omega_D t \end{cases}$$

```
def func(t, y, g, l, q, F_D, Omega_D):
    return [
        y[1],
        -(g/l)*y[0] - q*y[1] + F_D*np.sin(Omega_D*t)
    ]
```

`y[0]` $\equiv \theta$ `y[1]` $\equiv \omega$

Jacobian for equations

- For many methods, convergence improves by specifying Jacobian

$$\begin{cases} f_1(\theta,\omega,t) = \omega \\ f_2(\theta,\omega,t) = -\dfrac{g}{l}\theta - q\omega + F_D \sin\Omega_D t \end{cases}$$

$$\begin{bmatrix} \dfrac{\partial f_1}{\partial \theta} & \dfrac{\partial f_1}{\partial \omega} \\ \dfrac{\partial f_2}{\partial \theta} & \dfrac{\partial f_2}{\partial \omega} \end{bmatrix}$$

```
def jac(t, y, g, l, q, F_D, Omega_D):
    return [
        [0.0, 1.0],
        [-g/l, -q]
    ]
```

Integrate ODEs

- Integrate from `t0` to `t_max` in steps `delta_t`

integration method RKF(4, 5)

```
from scipy.integrate import ode
…
ode_sys = ode(func, jac).set_integrator('dopri5')
ode_sys.set_initial_value([theta0, omega0], t0)
ode_sys.set_f_params(g, l, q, F_D, Omega_D)
ode_sys.set_jac_params(g, l, q, F_D, Omega_D)

while ode_sys.successful() and ode_sys.t < t_max:
    ode_sys.integrate(ode_sys.t + delta_t)
    print(ode_sys.t, ode_sys.y[0], ode_sys.y[1])
```

Signal processing

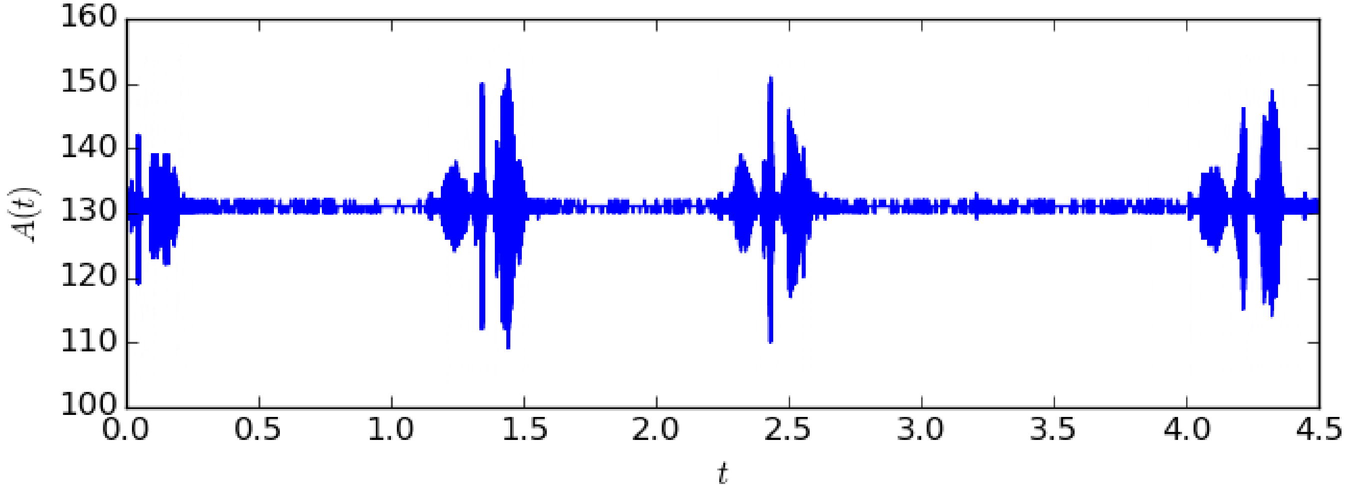
- Remove noise from sound file (WAV)
  - Read WAV file

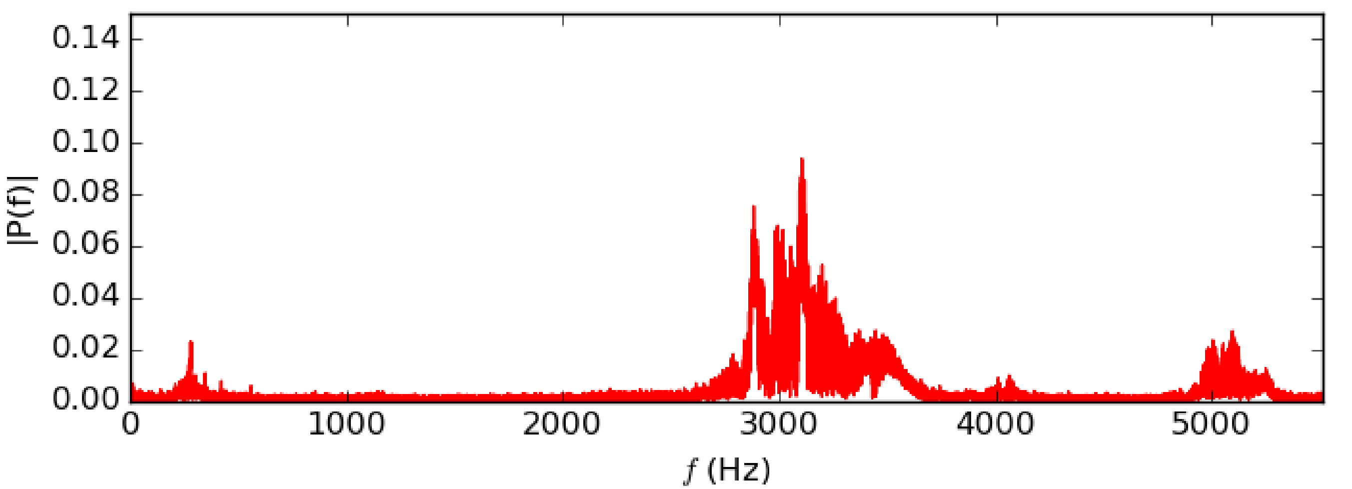
```
…
from scipy.io import wavfile
…
sample_rate, signal = wavfile.read(wav_file_name)
```

  - Perform FFT to compute frequency spectrum

```
…
n = len(signal)
freq = sample_rate*np.arange(n)/n
Y = sp.fft(signal)/n
```

Original signal





Create highpass filter

- Import signal processing package

```
…
import scipy.signal
…
```

- Create IIR digital filter

fraction of Nyquist frequency

order of the filter

minimum attenuation

filter type

IIR filter type

```
…
b, a = sp.signal.iirfilter(17, cutoff,
                           rs=min_attenuation,
                           btype='highpass',
                           analog=False,
                           ftype='cheby2')
…
```

Filter signal

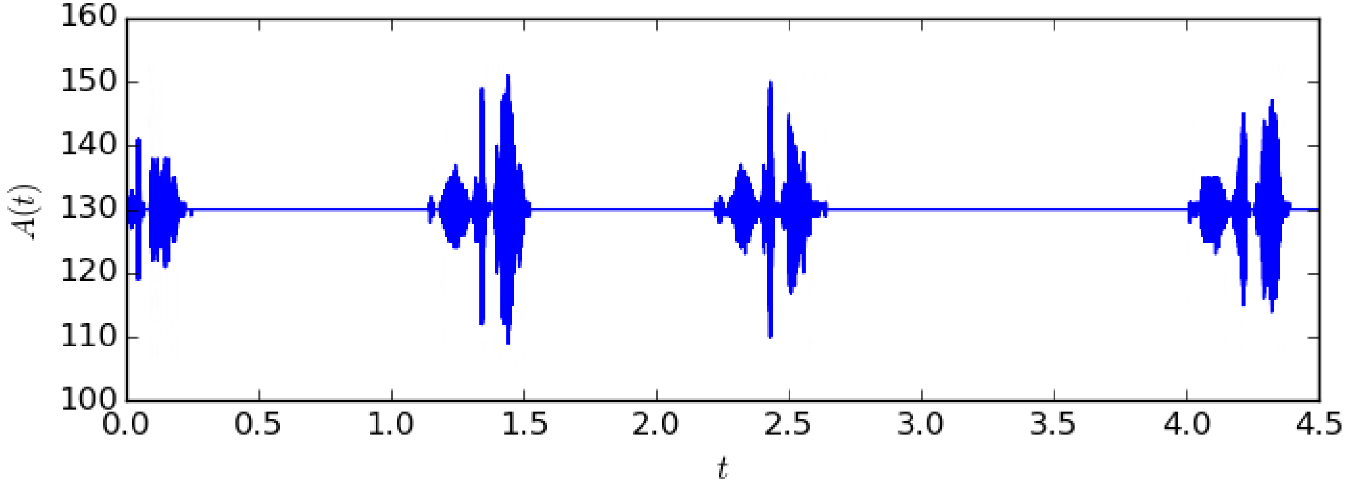
- Apply filter

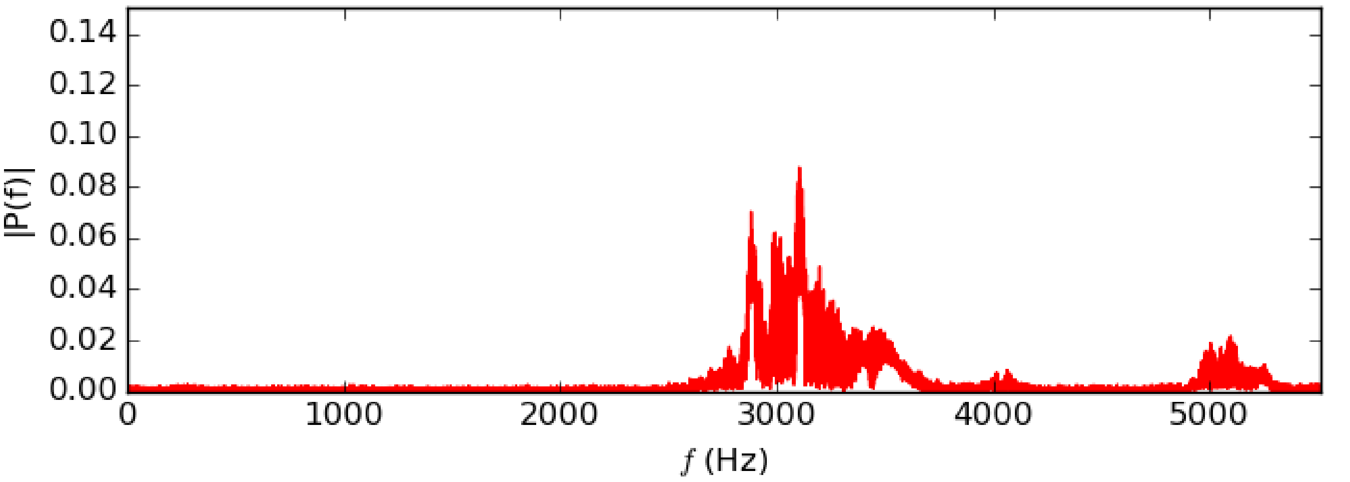
```
…
base = np.uint8(np.mean(signal))
filtered_signal = sp.signal.filtfilt(b, a, signal)
wav_signal = base + np.array(filtered_signal,
                             dtype=np.uint8)
…
```

- Write signal to WAV file

```
…
wavfile.write(filtered_wav_file_name, sample_rate,
              wav_signal)
…
```

Filtered signal





Code Pack 26

A. Scipy: See the code