Array Computing and Curve Plotting

Mojtaba Alaei

April 9, 2023

Content of the course

Using Lists for Collecting Function Data

```
>>>def f(x):
...     return x**3  # sample function
...
>>>n = 5 # no of points along the x axis
>>>dx = 1.0/(n-1) # spacing between x points in [0,1]
>>>xlist = [i*dx for i in range(n)]
>>>ylist = [f(x) for x in xlist]
>>>pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Lists are flexible

```
mylist = [2, 6.0, 'tmp.ps', [0,1]]
```

In cases where the elements are of the same type and the number of elements is fixed, **arrays** can be used instead. The benefits of arrays are faster computations, less memory demands, and extensive support for mathematical operations on the data

Basics of Numerical Python Arrays

```
import numpy as np
```

To convert a list r to an array:

```
a = np.array(r)
```

To create a new array of length n:

```
a = np.zeros(n)
```

An array to have n elements with uniformly distributed values in an interval [p, q]:

```
a = np.linspace(p, q, n)
```

a[1:-1] picks out all elements except the first and the last, but contrary to lists, a[1:-1] is not a copy of the data in a:

```
b = a[1:-1]
b[2] = 0.1
```

will also change a[3] to 0.1

Basics of Numerical Python Arrays

a[i:j:s] ?
a[0:-1:2] ?
a[::4] ?

Vectorization

Vectorization:

Operations on whole arrays, instead of using Python for loops, is called vectorization and is very convenient and very efficient (and an important programming technique to master)

```
>>>x = np.linspace(0, 1, n)
>>>y = np.zeros(n)
>>>for i in range(n):
>>>...y[i] = f(x[i])
```

Instead of using loops, we can use vectrorization in numpy:

```
>>> y = f(x)
```

Vectorization

Use numpy for sin, exp, ... functions instead of math:

```
from math import sin, cos, exp
import numpy as np
r = np.zeros(len(x))
for i in range(len(x)):
    r[i] = sin(x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

So to use vectorization we replace loop in the following way:

```
r = np.sin(x)*np.cos(x)*np.exp(-x**2) + 2 + x**2
```

Or

```
from numpy import sin, cos, exp
r = sin(x)*cos(x)*exp(-x**2) + 2 + x**2
```

Vectorization Time

```
import numpy as np
from time import time
def f(x):
    return x**3
n=1000000
x=np.linspace(1,1000,n)
y=np.zeros(n)

t0=time()
for i in range(n):
    y[i]=f(x[i])
print("Time without vectorization=", time()-t0)

t0=time()
y=f(x)
print("Time with vectorization=",time()-t0)
```

Vectorization error when we use branching in functions

```
import numpy as np
def f(x):
    if x> 0:
      y=np.sin(x)
    else:
      y=0.0
    return y
x=np.linspace(-np.pi, np.pi,100)
y=f(x)
```

Vectorization error when we use branching in functions

```
import numpy as np
def f(x):
    if x> 0:
      y=np.sin(x)
    else:
      y=0.0
    return y
x=np.linspace(-np.pi, np.pi,100)
y=f(x)
```

```
import numpy as np
@np.vectorize # this line is needed
def f(x):
    if x> 0:
      y=np.sin(x)
    else:
      y=0.0
    return y
```

Copying Arrays

Let x be an array. The statement a = x makes a refer to the same array as x. Changing a will then also affect x:

```
>>> import numpy as np
>>> x = np.array([1, 2, 3.5])
>>> a = x
>>> a[-1] = 3 # this changes x[-1] too!
>>> x
array([ 1., 2., 3.])
```

Changing a without changing x requires a to be a copy of x:

```
>>> a = x.copy()
>>> a[-1] = 9
>>> a
array([ 1., 2., 9.])
>>> x
array([ 1., 2., 3.])
```

In-Place Arithmetics

The difference between a = a+b and a += b

- In the statement a = a + b, the sum a + b is first computed, yielding a new array, and then the name a is bound to this new array.
- In the statement a += b, elements of b are added directly into the elements of a (in memory). There is no hidden intermediate array as in a = a + b. This implies that a += b is more efficient than a = a + b since Python avoids making an extra array. We say that the operators +=, *=, and so on, perform in-place arithmetics in arrays.

In-Place Arithmetics

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

The computation actually goes as follows with seven hidden arrays for storing intermediate results:

```
1. r1 = x**4
2. r2 = 3*r1
3. r3 = 2*x
4. r4 = r2 + r3
5. r5 = r4 + 4
6. r6 = x + 1
7. r7 = r5/r6
8. a = r7
```

With in-place arithmetics we can get away with creating three new arrays, at a cost of a significantly less readable code:

```
a= x.copy()
a**= 4
a*= 3
a+= 2*x
a+= 4
a/= x + 1
```

Allocating Arrays

To make an array with the size and the type of another existing array:

```
a = x.copy()
```

Or

```
a = np.zeros(x.shape, x.dtype)
```

To ensure that an object is an array:

```
a = np.asarray(a)
```

Generalized Indexing

Index slicing is the technical name for the syntax M[lower:upper:step] to extract part of an array. The slice lower:upper:step corresponds to the index set lower, lower+step, lower+2*step, ... Such an index set can be given explicitly too: a[range(f,t,i)]:

```
>>> a = np.linspace(1, 8, 8)
>>> a
array([ 1., 2., 3., 4., 5., 6., 7., 8.])
>>> a[[1,6,7]] = 10
>>> a
array([ 1., 10., 3., 4., 5., 6., 10., 10.])
>>> a[range(2,8,3)] = -2 # same as a[2:8:3] = -2
>>> a
array([ 1., 10., -2., 4., 5., -2., 10., 10.])
```

We can also use boolean arrays to generate an index set:

```
>>> a[a < 0] # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([ 1., 10., 10., 4., 5., 10., 10., 10.])
>>> # Replace elements where a is 10 by the first
>>> # elements from another array/list:
>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]
>>> a
array([ 1., 10., 20., 4., 5., 30., 40., 50.])
```

Find the index of value in Numpy Array

```
>>> a=np.array([2,3,53,12,21,32,9])
>>> np.where(a==2)
(array([0]),)
>>> np.where(a==53)
(array([2]),
>>> np.where(a>12)
(array([2, 4, 5]),)
```

Array Type: The N-dimensional array (ndarray)

A numerical Python array has a tpye which name is ndarray:

```
>>> a = np.linspace(-1, 1, 3)
>>> a
array([-1., 0., 1.])
>>> type(a)
<type 'numpy.ndarray'>
```

To learn about ndarray see the following link:
docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

Difference between arange and linspace function

```
>>> a=np.linspace(0,5,10)  # step = (5-0)/9
>>>
>>> b=np.arange(0,5,0.5) # step =0.5
>>>
>>> a
array([ 0.        ,  0.55555556,  1.11111111,  1.66666667,  2.22222222
        2.77777778,  3.33333333,  3.88888889,  4.44444444,  5.
])
>>> b
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
>>> len(a)
10
>>> len(b)
10
```

Shape Manipulation

The shape attribute in array objects holds the shape, i.e., the size of each dimension. A function size returns the total number of elements in an array.

```
>>> a = np.linspace(-1, 1, 6)
>>> a.shape
(6,)
>>> a.size
6
>>> a.shape = (2, 3)
>>> a.shape
(2, 3)
>>> a.size # total no of elements
6
>>> a.shape = (a.size,) # reset shape
>>> a = a.reshape(3, 2) # alternative
>>> len(a) # no of rows
3
```

Two-Dimensional Numerical Python Arrays

```
>>> Cdegrees = [-30 + i*10 for i in range(3)]
>>> Fdegrees = [9./5*C + 32 for C in Cdegrees]
>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
>>> print(table)
[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
```

To convert to the an array:

```
>>> table2 = np.array(table)
>>> print(table2)
[[-30. -22.]
[-20. -4.]
[-10. 14.]]
>>> type(table2)
<type 'numpy.ndarray'>
```

Two-Dimensional Numerical Python Arrays

Access to an element in list:

```
>>> table[1][0] # table[1] is [-20,4], whose index 0 holds -20
-20
```

This syntax works for two-dimensional arrays too:

```
>>> table2[1][0]
-20.0
```

But there is another syntax which is more common for arrays:

```
>>> table2[1,0]
-20.0
```

Two-Dimensional Numerical Python Arrays

```
from numpy import empty, zeros
# Create arrays
# Creat an array with (M+1)*(M+1) zero elements
V1 = zeros([M+1, M+1], float)
# Creat a empty array with (M+1)*(M+1) dimension
V2 = empty([M+1, M+1], float)
```

Arrays products in numpy

```
>>> import numpy as np
>>> a=np.array([1,2,3])
>>> b=np.array([1,2,3])
>>> a*b
array([1, 4, 9])
```

Matrix Objects

NumPy also has a matrix type called matrix or mat for one- and two-dimensional arrays:

```
>>> import numpy as np
>>> x1 = np.array([1, 2, 3], float)
>>> x2 = np.matrix(x1)                    # or mat(x1)
>>> x2                                    # row vector
matrix([[ 1., 2., 3.]])
>>> x3 = x2.transpose()                   # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> isinstance(x3, np.matrix)
True
```

Matrix Objects

The matrix-matrix, vector-matrix, or matrix-vector product:

```
>>> A = eye(3)          # identity matrix, also we can use identity(3)
>>> A
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])
>>> A = mat(A)
>>> A
matrix([[ 1., 0., 0.],
        [ 0., 1., 0.],
        [ 0., 0., 1.]])
>>> y2 = x2*A                # vector-matrix product
>>> y2
matrix([[ 1., 2., 3.]])
>>> y3 = A*x3                # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

Matrix Objects

One should note here that the multiplication operator between standard ndarray objects is quite different:

```
>>> A*x1                    # no matrix-array product!
Traceback (most recent call last):
...
ValueError: matrices are not aligned

>>> # try array*array product:
>>> A = (zeros(9) + 1).reshape(3,3)
>>> A
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.]])
>>> A*x1                        # [A[0,:]*x1, A[1,:]*x1, A[2,:]*x1]
array([[ 1., 2., 3.],
       [ 1., 2., 3.],
       [ 1., 2., 3.]])
>>> B = A + 1
>>> A*B                         # element-wise product
array([[ 2., 2., 2.],
       [ 2., 2., 2.],
       [ 2., 2., 2.]])
>>> A = mat(A); B = mat(B)
>>> A*B                         # matrix-matrix product
matrix([[ 6., 6., 6.],
        [ 6., 6., 6.],
        [ 6., 6., 6.]])
```

dot product of arrays

```
>>> A=np.array([[1,2],[3,4]])
>>> B=np.array([[2,4],[4,4]])
>>> A*B
array([[ 2,  8],
       [12, 16]])
>>> A
array([[1, 2],
       [3, 4]])
>>> B
array([[2, 4],
       [4, 4]])
>>> np.dot(A,B)  # it is matrix product
array([[10, 12],
       [22, 28]])
>>> Am=np.matrix(A)
>>> Bm=np.matrix(B)
>>> Am*Bm
matrix([[10, 12],
        [22, 28]])
```

dot product of arrays

np.dot can act as dot product of two vectors:

```
>>> a=np.array([1,2,3])
>>> b=np.array([2,4,3])
>>> np.dot(a,b)
19
```

np.dot can use also as follows:

```
>>> a.dot(b)
19
>>> A=np.array([[1,2],[3,4]])
>>> B=np.array([[2,4],[4,4]])
>>> C=np.array([[1,3],[3,5]])
>>> A.dot(B).dot(C)
array([[ 46,  90],
       [106, 206]])
>>> Am=np.matrix(A)
>>> Bm=np.matrix(B)
>>> Cm=np.matrix(C)
>>> Am*Bm*Cm
matrix([[ 46,  90],
        [106, 206]])
```

eye, diag,ones

```
>>> np.eye(5)  # 5x5 Identical matrix
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
>>> np.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> np.diag([4,2,3]) # 3x3  diagonal matrix
array([[4, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>>
```

eye, diag, ones

```
>>> np.diag([10,2,4],1)
array([[ 0, 10,  0,  0],
       [ 0,  0,  2,  0],
       [ 0,  0,  0,  4],
       [ 0,  0,  0,  0]])
>>> np.diag([10,2,4],-1)
array([[ 0,  0,  0,  0],
       [10,  0,  0,  0],
       [ 0,  2,  0,  0],
       [ 0,  0,  4,  0]])
>>>
```

Also we can pick diagonal element with diag:

```
>>> A=np.array([[2,3,5],[1,4,1],[2,4,1]])
>>> A
array([[2, 3, 5],
       [1, 4, 1],
       [2, 4, 1]])
>>> np.diag(A)
array([2, 4, 1])
```

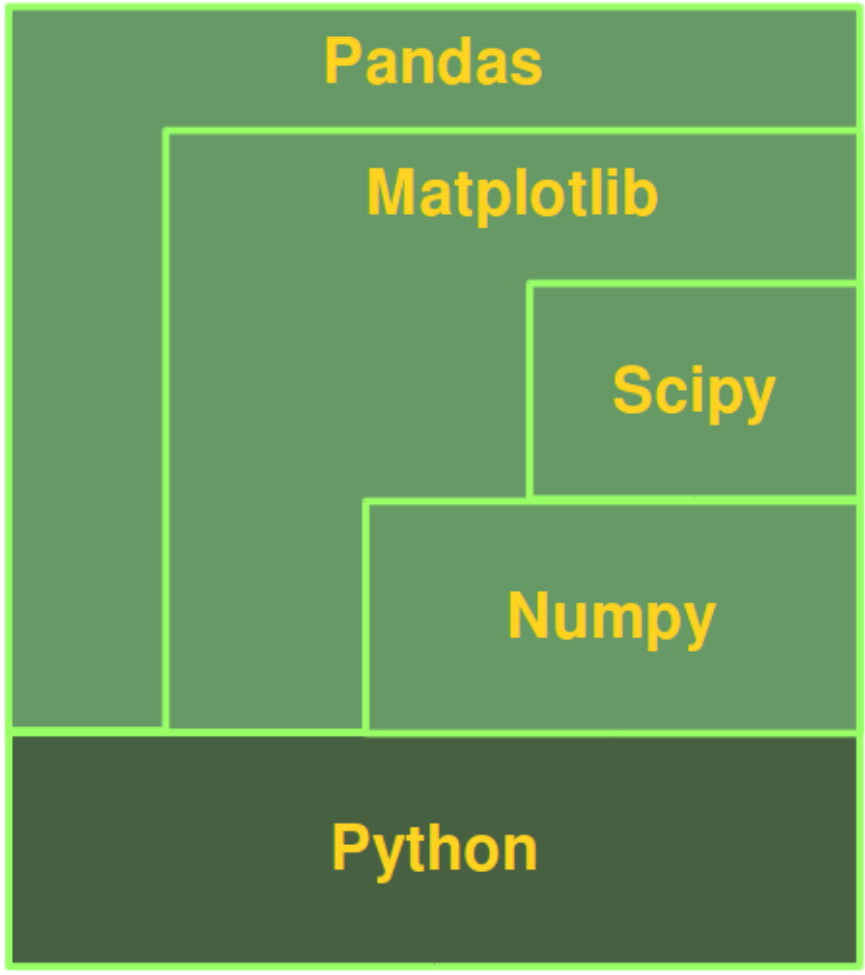
numpy.random

```
>>> np.random.rand()
0.14176743230683708
>>> np.random.rand(4,4)
array([[ 0.65830469,  0.04210806,  0.7115102 ,  0.02581672],
       [ 0.39467673,  0.47257193,  0.98651384,  0.00791865],
       [ 0.66119377,  0.80399312,  0.0777308 ,  0.14450883],
       [ 0.27636948,  0.04903494,  0.27021341,  0.55847552]])
>>>
```

Summary

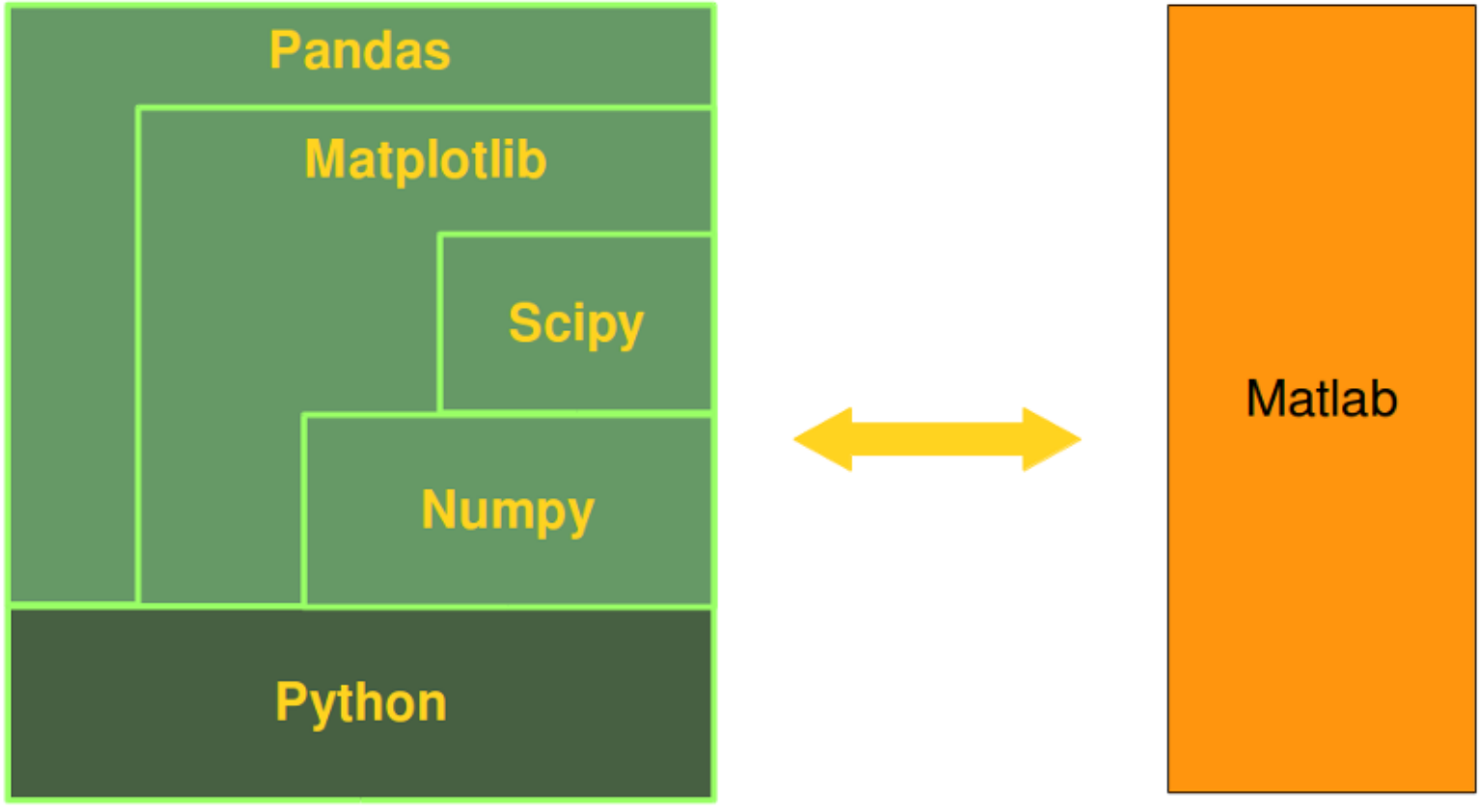
`array(ld)`	copy **list** data ld to a numpy array
`asarray(d )`	make array of data d (copy **if list**, no copy **if** already array)
`zeros(n)`	make a **float** vector/array of length n, with
`zeros(n, int)`	make an **int** vector/array of length n with ze
`zeros((m,n))`	make a two-dimensional **float** array with shap
`zeros(x.shape, x.dtype)`	make array of same shape as x **and** same element data **type**
`linspace(a,b,m)`	uniform sequence of m numbers between a **and** (b **is** included **in** the sequence)
`a.shape`	**tuple** containing a's shape
`a.size`	total no of elements in a
`len(a)`	length of a one-dim. array a (same as a.shap
`a.reshape(3,2)`	return a reshaped as 2 x 3 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices 1,. . . ,
`a[1:8:3]`	slice: reference data with indices 1, 4,. .
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	numpy functions applicable to arrays
`c = concatenate(a, b)`	c contains a with b appended
`c = where(cond, a1, a2)`	c[i] = a1[i] if cond[i], else c[i] = a2[i]
`isinstance(a, ndarray)`	is True if a is an array

Connections between Python, Numpy, Matplotlib, Scipy ...



https://www.python-course.eu/numerical_programming_with_python.php

Python, An Alternative to Matlab



https://www.python-course.eu/numerical_programming_with_python.php

Diagonalization using Linear algebra (numpy.linalg)

```
>>> import numpy.linalg  as la
>>> M=np.array([[2,0,0],[0,3,4],[0,4,9]])
>>> la.det(M)
21.999999999999996
>>> val, vec=la.eig(M)
>>> val # eigvalues
array([ 11.,   1.,   2.])
>>> vec
array([[ 0.        ,  0.        ,  1.        ],
       [ 0.4472136 ,  0.89442719,  0.        ],
       [ 0.89442719, -0.4472136 ,  0.        ]])

>>> vec[:,0] # first eigenvector
array([ 0.        ,  0.4472136 ,  0.89442719])
>>> vec[:,1] # second eigenvector
array([ 0.        ,  0.89442719, -0.4472136 ])
>>> vec[:,2] #third
array([ 1.,  0.,  0.])
```

Quiz1:spectral decomposition

Check if M is equal to its spectral decomposition:

$$M = \sum_i \lambda_i |\lambda_i\rangle\langle\lambda_i| \tag{1}$$

Quiz2

The matrix file "n_mmp_mat" is a density matrix with two 7×7 matrix with complex elements for spin up and down (Don't worry, you do not need to know what density matrix is). The file contains 7 columns and 28 rows. Each two numbers in a row shows a complex number (first one is real part and the second one is imaginary part). So the first 14 rows make a 7×7 comlex matrix (for spin up) and anthor 14 rows make an another 7×7 comlex matrix (for spin down).

Quiz:

By using numpy (you need to know about "loadtxt", "linalg.eig"), calculate eigenvectors and eigenvalues of the first "7×7 comlex matrix"

Solve linear systems with np.solve

(this part is adapted from https://riptutorial.com/numpy)
Consider the following three equations:

```
x0 + 2 * x1 + x2 = 4
          x1 + x2 = 3
x0 +           x2 = 5
```

We can express this system as a matrix equation A * x = b with:

```
A = np.array([[1, 2, 1],
              [0, 1, 1],
              [1, 0, 1]])
b = np.array([4, 3, 5])
```

Then, use np.linalg.solve to solve for x:

```
x = np.linalg.solve(A, b)
# Out: x = array([ 1.5, -0.5,  3.5])
```

Solve linear systems with np.solve

A must be a square and full-rank matrix: All of its rows must be be linearly independent. A should be invertible/non-singular (its determinant is not zero). For example, If one row of A is a multiple of another, calling linalg.solve will raise LinAlgError: Singular matrix:

```
A = np.array([[1, 2, 1],
              [2, 4, 2],   # Note that this row 2 * the first row
              [1, 0, 1]])
b = np.array([4,8,5])
```

Such systems can be solved with np.linalg.lstsq.

Least squares solution to a linear system : np.linalg.lstsq

(this part is adapted from https://riptutorial.com/numpy)
Consider the four equations:

```
x0 + 2 * x1 + x2 = 4
x0 + x1 + 2 * x2 = 3
2 * x0 + x1 + x2 = 5
x0 + x1 + x2 = 4
```

We can express this as a matrix multiplication A * x = b:

```
A = np.array([[1, 2, 1],
              [1,1,2],
              [2,1,1],
              [1,1,1]])
b = np.array([4,3,5,4])
```

Least squares solution to a linear system : np.linalg.lstsq

Then solve with np.linalg.lstsq:

```
x, residuals, rank, s = np.linalg.lstsq(A,b)
```

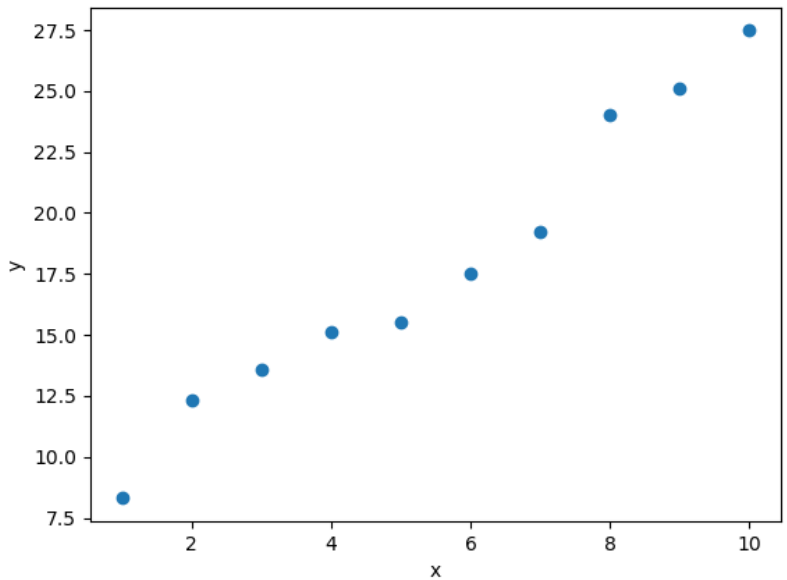
- x is the solution,
- residuals is sum of squared residuals; Squared Euclidean 2-norm for each column in b - A * x,
- rank the matrix rank of input A,
- and s the singular values of A,

For the above example, residuals can be calculated by the following equation:

```
r=0
for i in range(4):
    r=r+(np.dot(x, A[i,:])-b[i])**2
print(r)
```

solving a linear equation

Consider the following data:



To find the best fit line, we should find θ_1 and θ_0 (the slope and intercept) to minimize the following equation:

$$|\theta_1 x + \theta_0 - y|^2$$

solving a linear equation

So for m data, we want to satisfy the following equations as much as possible:

$$\begin{aligned} y^{(1)} &= \theta_1 x^{(1)} + \theta_0 \\ y^{(2)} &= \theta_1 x^{(2)} + \theta_0 \\ y^{(3)} &= \theta_1 x^{(3)} + \theta_0 \\ \vdots \quad &\vdots \quad \vdots \\ y^{(m)} &= \theta_1 x^{(m)} + \theta_0 \end{aligned}$$

Therefore, we should minimize the difference between $y^{(i)}$ and $\theta_1 x^{(i)} + \theta_0$ (i.e. least squares solution):

$$\min |\theta_1 x + \theta_0 - y|^2 = \min \sum_{i=1}^{m} (\theta_1 x^{(i)} + \theta_0 - y^{(i)})^2$$

Here θ_1 and θ_0 are unkonw. How do we find θ_1 and θ_0 using np.linalg.lstsq?

solving a linear equation

Find θ_1 and θ_0 using np.linalg.lstsq:
Rewrite the linear equation:

$$\begin{bmatrix} x^{(1)} & 1 \\ x^{(2)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix} \times \begin{bmatrix} \theta_1 \\ \theta_0 \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

These equations are similar to $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} x^{(1)} & 1 \\ x^{(2)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} \theta_1 \\ \theta_0 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

So we can use np.linalg.lstsq to find θ_1 and θ_0.

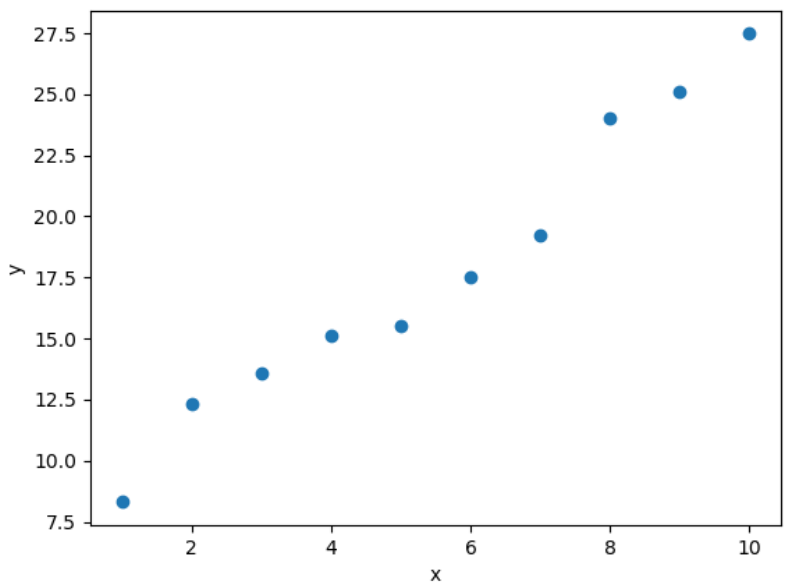
solving a linear equation; example

Create fake linear data:

```
np.random.seed(42)

x=np.linspace(1,10,10)
y=2*x+5+3.5*np.random.rand(len(x))

plt.plot(x,y,'o')
plt.xlabel("x")
plt.ylabel("y")
```



solving a linear equation; example

Creat **A**, **b**:

```
A=np.hstack([np.c_[x],np.c_[np.ones(len(x))]])
print(A)
```

```
[[ 1.  1.]
 [ 2.  1.]
 [ 3.  1.]
 [ 4.  1.]
 [ 5.  1.]
 [ 6.  1.]
 [ 7.  1.]
 [ 8.  1.]
 [ 9.  1.]
 [10.  1.]]
```

```
b=np.c_[y]
print(b)
```

```
[[ 3.31089042]
 [ 7.32750007]
 [ 8.5619788 ]
 [10.09530469]
 [10.54606524]
 [12.54598082]
 [14.20329264]
 [19.03161651]
 [20.10390254]
 [22.47825402]]
```

solving a linear equation; example

Solving $\mathbf{Ax} = \mathbf{b}$:

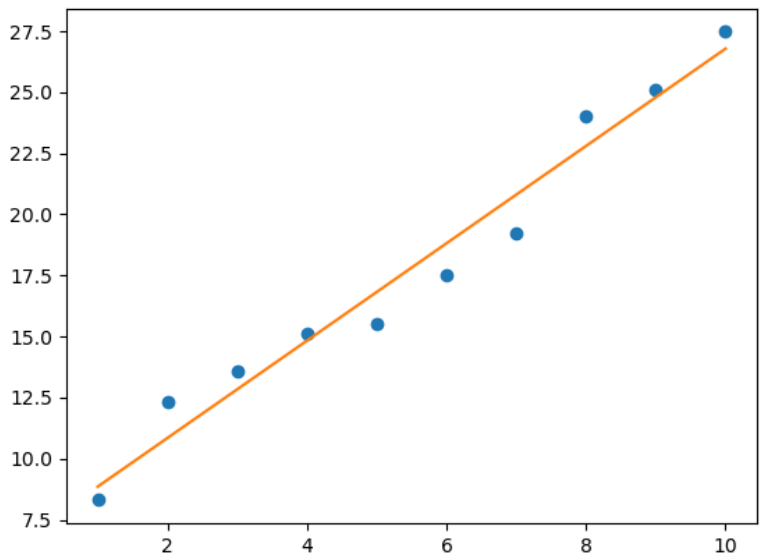
```
theta=np.linalg.lstsq(A,b)[0]
print(theta)
```

```
[[1.9915949 ]
 [6.86670665]]
```

solving a linear equation; example

Plot the line:

```
ypredict= theta[0][0]*x+theta[1][0]
plt.plot(x,y,'o')
plt.plot(x,ypredict)
```



extend to the multilinear equations

$$
\begin{aligned}
y^{(1)} &= \theta_1 x_1^{(1)} + \theta_2 x_2^{(1)} + \cdots + \theta_0 \\
y^{(2)} &= \theta_1 x_1^{(2)} + \theta_2 x_2^{(2)} + \cdots + \theta_0 \\
y^{(3)} &= \theta_1 x_1^{(3)} + \theta_2 x_2^{(3)} + \cdots + \theta_0 \\
\vdots &\quad \vdots \quad \vdots \\
y^{(m)} &= \theta_1 x_1^{(m)} + \theta_2 x_2^{(m)} + \cdots + \theta_0
\end{aligned}
\tag{2}
$$

$$
\begin{bmatrix}
x_1^{(1)} & x_2^{(1)} & \cdots & 1 \\
x_1^{(2)} & x_2^{(2)} & \cdots & 1 \\
\vdots & \vdots & \vdots & \vdots \\
x_1^{(m)} & x_2^{(m)} & \vdots & 1
\end{bmatrix}
\times
\begin{bmatrix}
\theta_1 \\ \theta_2 \\ \vdots \\ \theta_0
\end{bmatrix}
=
\begin{bmatrix}
y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)}
\end{bmatrix}
$$

Further reading

- https://github.com/jrjohansson/scientific-python-lectures
- http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scienti
  python-lectures/master/Lecture-2-Numpy.ipynb
- http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html
- http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html
- http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix
- http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

Curve Plotting: matplotlib.pyplot

```
import matplotlib.pyplot as plt
import numpy as np
def f(t):
    return t**2*np.exp(-t**2)
t = np.linspace(0, 3, 51)          # 51 points between 0 and 3
y = np.zeros(len(t))               # allocate y with float elements
y = f(t)                           # vectorization
plt.plot(t, y)
plt.show()
```

matplotlib.pyplot: title, legend, axis, savefig

```
plot(t, y)
xlabel('t')
ylabel('y')
legend(['t^2*exp(-t^2)'])
axis([0, 3, -0.05, 0.6])          # [tmin, tmax, ymin, ymax]
title('My First Matplotlib Demo')
savefig('tmp1.eps')               # produce PostScript
savefig('tmp1.png')               # produce PNG
show()
```

Plotting Multiple Curves

```
import  matplotlib.pyplot as plt
import numpy as np

def f1(t):
    return t**2*np.exp(-t**2)
def f2(t):
    return t**2*f1(t)
t = np.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plt.plot(t, y1, 'r-')
plt.plot(t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Plotting two curves in the same plot')
plt.show()
```

In these plot commands, we have also specified the line type: r- means red (r) line (-), while bo means a blue (b) circle (o) at each data point.

Matplotlib colors

Using single letter:

- b: blue
- g: green
- r: red
- c: cyan
- m: magenta
- y: yellow
- k: black
- w: white

Matplotlib colors

Using RGB:

```
import matplotlib.pyplot as plt
import numpy as np

x=np.linspace(0,2*np.pi)
plt.title("Using RGB for color")
plt.xlabel("x")
plt.ylabel("y")
plt.plot(x,np.sin(x),     color=(1.0, 0.0 ,0.0),
         label='Red',    lw=4.0)
plt.plot(x,np.cos(x),     color=(0.0, 1.0 ,0.0),
         label='Green', lw=4.0)
plt.plot(x,np.cos(x-0.5),color=(0.0, 0.0 ,1.0),
         label='Blue',  lw=4.0 )
plt.plot(x,np.sin(x-0.5),color=(0.7, 0.25,0.25),
         label='0.7R+0.25G+0.25B',
         lw=4.0)
plt.legend()
plt.show()
```

Placing Several Plots in One Figure

```
import matplotlib.pyplot as plt
import numpy as np

def f1(t):
    return t**2*np.exp(-t**2)
def f2(t):
    return t**2*f1(t)

plt.figure() # make separate figure
plt.subplot(2, 1, 1)
t = np.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)
plt.plot(t, y1, 'r-', t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Top figure')
plt.subplot(2, 1, 2)
t3 = t[::4]
y3 = f2(t3)
plt.plot(t, y1, 'b-', t3, y3, 'ys')
plt.xlabel('t')
plt.ylabel('y')
plt.axis([0, 4, -0.2, 0.6])
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.title('Bottom figure')
plt.savefig('tmp4.eps')
plt.show()
```

Matplotlib; Pyplot

The Matplotlib developers do not promote the matplotlib.pylab interface. Instead, they recommend the matplotlib.pyplot module and prefix Numerical Python and Matplotlib functionality by short forms of their package names:

```
import numpy as np
import matplotlib.pyplot as plt
```

The commands in matplotlib.pyplot are similar to those in matplotlib.pylab.

```
plt.plot(t, y)
plt.legend(['t^2*exp(-t^2)'])
plt.xlabel('t')
plt.ylabel('y')
plt.axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
plt.title('My First Matplotlib Demo')
plt.show()
plt.savefig('tmp2.eps')      # produce PostScript
```

Plotting data in a file

suppose we have a file,ld1.wfc, with 5 columns of data

```
#          r              3 S             2 P             2 S             1 S
  0.001831563    0.005941348    0.000094305    0.025734199    0.109245136
  0.001877930    0.006088921    0.000099117    0.026373394    0.111958793
  0.001925470    0.006240086    0.000104175    0.027028146    0.114738511
  0.001974213    0.006394925    0.000109489    0.027698818    0.117585831
  0.002024191    0.006553526    0.000115074    0.028385779    0.120502323
  0.002075433    0.006715974    0.000120943    0.029089406    0.123489593
  0.002127973    0.006882359    0.000127111    0.029810086    0.126549278
  0.002181843    0.007052772    0.000133592    0.030548211    0.129683050
  0.002237077    0.007227305    0.000140403    0.031304183    0.132892613
  0.002293709    0.007406052    0.000147560    0.032078411    0.136179710
  0.002351774    0.007589111    0.000155080    0.032871312    0.139546114
  0.002411310    0.007776579    0.000162983    0.033683312    0.142993637
  0.002472352    0.007968557    0.000171287    0.034514847    0.146524128
  0.002534940    0.008165147    0.000180013    0.035366358    0.150139468
  0.002599112    0.008366453    0.000189182    0.036238298    0.153841579
  0.002664909    0.008572582    0.000198816    0.037131127    0.157632419
  0.002732372    0.008783642    0.000208940    0.038045313    0.161513983
  0.002801542    0.008999742    0.000219576    0.038981335    0.165488305
  0.002872463    0.009220996    0.000230753    0.039939679    0.169557457
  0.002945180    0.009447518    0.000242496    0.040920842    0.173723551
  0.003019738    0.009679425    0.000254834    0.041925330    0.177988736
  0.003096183    0.009916835    0.000267798    0.042953655    0.182355202
  0.003174563    0.010159869    0.000281418    0.044006342    0.186825177
  0.003254928    0.010408651    0.000295728    0.045083923    0.191400931
  0.003337326    0.010663305    0.000310763    0.046186940    0.196084773
  0.003421811    0.010923958    0.000326558    0.047315946    0.200879052
  0.003508435    0.011190740    0.000343153    0.048471500    0.205786159
  0.003597251    0.011463783    0.000360587    0.049654174    0.210808522
  0.003688316    0.011743221    0.000378903    0.050864546    0.215948615
  0.003781686    0.012029189    0.000398145    0.052103206    0.221208947
  0.003877420    0.012321826    0.000418358    0.053370751    0.226592072
  0.003975578    0.012621271    0.000439593    0.054667791    0.232100582
  0.004076220    0.012927667    0.000461900    0.055994941    0.237737110
  0.004179410    0.013241160    0.000485332    0.057352828    0.243504331
```

Plotting data in a file

By using "loadtxt" we can convert the data to the array:

```
from numpy import loadtxt
from matplotlib.pyplot import plot,xlabel,ylabel,xlim,show, legend
r, S3, P2, S2, S1  = loadtxt('ld1.wfc', unpack = True)
xlabel("r")
ylabel("Psi(r)")
xlim(0,10)
plot(r,S3,"g--")
plot(r,P2,"r--")
plot(r,S2,"b--")
plot(r,S1,"y")
legend(('3S', '2P', '2S', '1S'), shadow=True )
show()
```

Plotting data in a file

We use "unpack=True", to separate data in to columns: r, S3, P2, S2, S1.
We can directly load all data to one variable: "alldata= np.loadtxt("ld1.wfc")".
Then the first column is "alldata[:,0]", the second is "alldata[:,1]" and so on.

matplotlib; logarithmic plot

From pylab examples: log_demo.py

```
import numpy as np
import matplotlib.pyplot as plt

plt.subplots_adjust(hspace=0.4)
t = np.arange(0.01, 20.0, 0.01)

# log y axis
plt.subplot(221)
plt.semilogy(t, np.exp(-t/5.0))
plt.title('semilogy')
plt.grid(True)

# log x axis
plt.subplot(222)
plt.semilogx(t, np.sin(2*np.pi*t))
plt.title('semilogx')
plt.grid(True)

# log x and y axis
plt.subplot(223)
plt.loglog(t, 20*np.exp(-t/10.0), basex=2)
plt.grid(True)
plt.title('loglog base 2 on x')

# with errorbars: clip non-positive values
ax = plt.subplot(224)
ax.set_xscale("log", nonposx='clip')
ax.set_yscale("log", nonposy='clip')

x = 10.0**np.linspace(0.0, 2.0, 20)
y = x**2.0
plt.errorbar(x, y, xerr=0.1*x, yerr=5.0 + 0.75*y)
ax.set_ylim(ymin=0.1)
ax.set_title('Errorbars go negative')

plt.show()
```

Simple Idea for Animation using Matplotlib

- Using plt.cla() to clear data and plot (axes)
- Using plt.pasue() to make a delay between plots
- Using plt.axis() to fix axes (not to allow changing according to data)

```
import matplotlib.pyplot as plt


n=200
plt.gca().set_aspect('equal',adjustable='box')#equal axes aspect ratio
#move a dot alone a line
for i in range(n):
  plt.cla()   #clear axes and data
  plt.axis([-1, 1, -1, 1])  #xmin, xmax, ymin, ymax
  plt.plot(i/n,i/n,'ro')
  plt.pause(0.05)
plt.show()
```

Simple Idea for Animation using Matplotlib

Another example: move partical with including its path (trajectory)

```
import matplotlib.pyplot as plt
n=200
plt.gca().set_aspect('equal',adjustable='box')#equal axes aspect ratio
#move a dot alone a line including its trajectory
xtraj=[]
ytraj=[]
for i in range(n):
  plt.cla()  #clear axes and data
  plt.axis([-1, 1, -1, 1])  #xmin, xmax, ymin, ymax
  plt.plot(i/n,i/n,'ro')
  xtraj.append(i/n)
  ytraj.append(i/n)
  plt.plot(xtraj,ytraj,'-')
  plt.pause(0.05)
plt.show()
```

Moving to Object-Oriented Interface

we learned about making plots using a procedural method, e.g.:

```
plt.figure()
plt.subplot(1, 1, 1)
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
```

To allow more customization, we need to move to a more object-based way to make the plots. This method involves storing various elements of the of the plots in variables (these are objects in object-oriented terminology). The above example becomes:

```
fig = plt.figure()  # create a figure object
ax = fig.add_subplot(1, 1, 1)  # create an axes object in the figure
ax.plot([1, 2, 3, 4])
ax.set_ylabel('some numbers')
```

http://www.ster.kuleuven.be/~pieterd/python/html/plotting/advanced.html

Moving to Object-Oriented Interface

This method is more convenient for advanced plots. One of the biggest advantages of using this method is that it allows users to easily handle multiple figures/axes without getting confused as to which one is currently active. For example:

```
fig1 = plt.figure()
fig2 = plt.figure()
ax1 = fig1.add_subplot(1, 1, 1)
ax2 = fig2.add_subplot(2, 1, 1)
ax3 = fig2.add_subplot(2, 1, 2)
```

http://www.ster.kuleuven.be/~pieterd/python/html/plotting/advanced.html

Further reading

- https://matplotlib.org/tutorials/index.html
- https://matplotlib.org/gallery/index.html