USC Viterbi
School of Engineering

NUMERICAL PYTHON

EE 541 – UNIT 2B

DR. BRANDON FRANZKE

Spring 2023

SCIENTIFIC PROGRAMMING LANGUAGES

	Fortran	C	C++	MATLAB	Python
Performance	+	+	+	o	o
Object Oriented	-	-	+	-	+
Exceptions	-	-	+	-	+
Open Source	+	+	+	-	+
Easy to Learn	o+	o	o-	+	+

- Python provides a robust platform for most possible tasks
- Strategy
  - write parts of the code that take computational cycles in C/Fortran
  - write other parts in high-level language (visualization, I/O, preprocessing)
  - Note: for heavy computational programs, 99% of CPU time on small % of code

MATLAB VS PYTHON

- MATLAB and Python share (good ideas)
  - higher level scripting (but slow)
  - fast compiled libraries
- Differences are important
  - MATLAB intended for Engineering and Computational Science tasks and combines powerful computation with visualization
  - MATLAB is commercial
  - Python is a modern general-purpose language and is easier to extend
  - For *large* projects usually best to use Python for program "glue"

PYTHON MODULES

- Bundle functions, classes, and code into a reusable package (a "library")
- Can import any Python file as a module
- Optional `__main__` program to run on import

```
# mymod.py
def afunction():
    pass

print(f"Name is {__name__}")

# python3 mymod.py, prints: Name is __main__

# program.py
import mymod
# print: Name is mymod
```

CONDITIONAL LIBRARY

```
def afunction():
    pass

def test_for_afunction():
    print("Running test…")

if __name__ == "__main__":
    test_for_afunction()
else
    print("Setting up")
    # code to prepare if imported as a library
```

ADVANCED TECHNIQUE: LIST COMPREHENSION

- List comprehension
  - uses “set builder notation”
  - convenient to map from one list to another

```
>>> [ 2**k for k in range(10) ]
# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

>>> import math
>>> xs = [0.1 * j for j in range(5)]
>>> ys = [math.exp(x) for x in xs]
```

LIST COMPREHENSION

- Conditional List Comprehension
  - map with filter

```
>>> [k for k in range(10)]
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> [2*k for k in range(10) if k ** 2 > 16]
# [10, 12, 14, 16, 18]
```

ADVANCED TECHNIQUES: RECURSION

- program where function calls itself
- conditional terminate at “base” or “terminal” case

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)

>>> factorial(6)
720
>>> factorial(0)
1
```

```
def fib(n):
  if n == 1:
    return 1
  elif n == 2:
    return 1
  else
    return fib(n-2) + fib(n-1)

>>> fib(6)
8
```

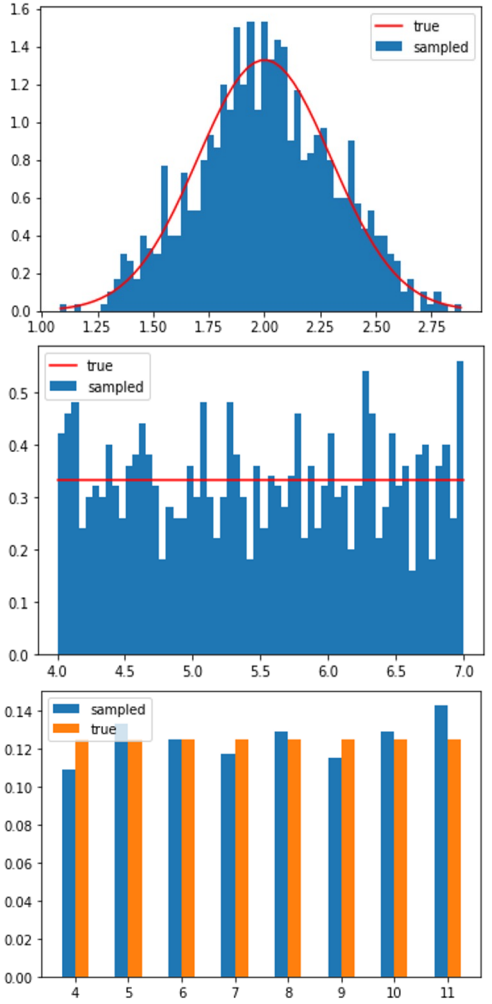
Random Numbers and Python

Built in python random module

- `random` module
- Sample (uniformly) integers, between 1-6:
  - random.randint(1, 6)
- Sample (uniformly) from a list of objects:
  - random.choice(["heads", "tails"])
- Sample from list with probability values:
  - random.choices(["rainy", "cloudy", "sunny", "foggy"], [0.1, 0.3, 0.5, 0.1])
- Shuffle a list:
  - random.shuffle(["heads", "tails", "monkey"])

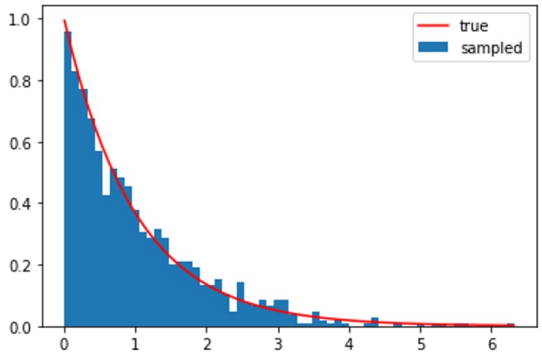
Randomness with numpy, sampling

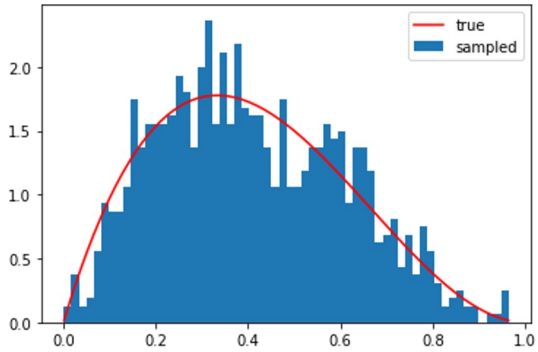
- Sample from list with probability:
  - numpy.random.choice(["rainy", "cloudy", "sunny", "foggy"], p = [0.1, 0.3, 0.5, 0.1])
- Sampling 1000 points from a Normal distribution, mu = 2, sigma = 0.3:
  - numpy.random.normal(2, 0.3, 1000)
- Sample 1000 points from a continuous uniformly distribution, a = 4, b = 7:
  - numpy.random.uniform(4, 7, 1000)
- Sample 1000 points from a discrete uniform distribution, a = 4, b = 12
  - numpy.random.randint(4, 12, 1000)



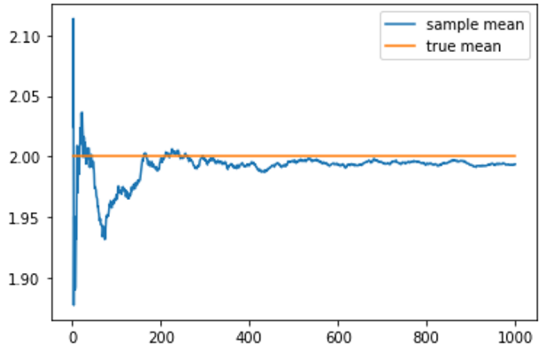
Randomness with numpy, sampling(cont.)

- Sampling 1000 points from an exponential distribution, lam = 1:
  - numpy.random.exponential(1, 1000)
- Sampling 1000 points from a Beta distribution, a = 2, b = 3:
  - numpy.random.beta(2, 3, 1000)



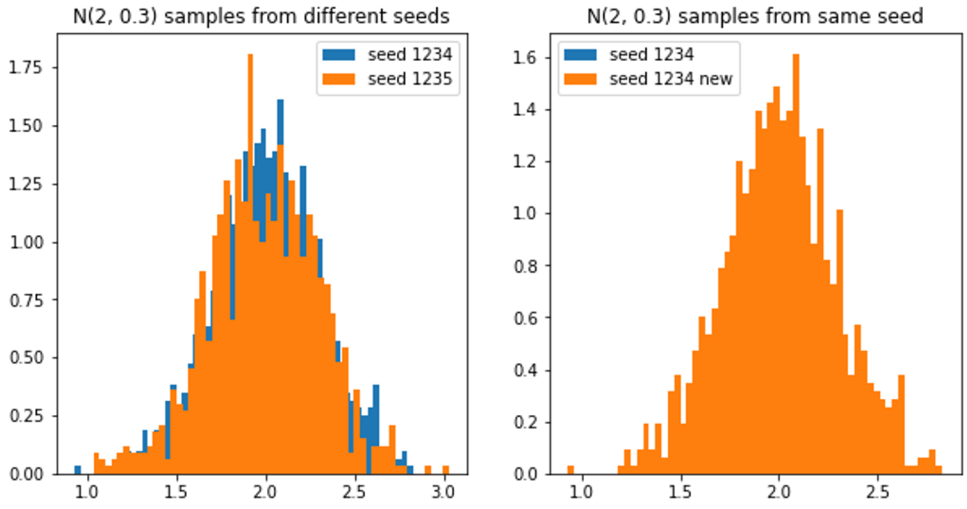


- Weak law of large numbers: Sample mean → Population mean:
  - samples = np.random.normal(2, 0.3, 1000)
  - sample_means = sample.cumsum()/np.arange(1,1001)



Randomness with numpy, seeds

- Can set seed to reproduce randomness:
  - np.random.seed(1234)
- Can set it on the built-in random module too:
  - random.seed(1234)
- Same seed generates same samples:
  - numpy.random.normal(2, 0.3, 1000)



NUMERIC COMPUTING

COMPUTATIONAL TASKS

- Processing data file, python and numpy (`array`)
- Generate random numbers (`numpy`)
- Linear algebra (`numpy`)
- Interpolation (`scipy.interpolation.interp`)
- Regression and curve-fit (`scipy.optimize.curve_fit`)
- Numerical integration (`scipy.integrate.quad`)
- Numerical ODE integration (`scipy.integrate.odeint`)

COMPUTATIONAL TASKS

- Root finding
  - (`scipy.optimize.fsolve, scipy.optimize.brentq`)
- Function minimization (`scipy.optimize.fmin`)
- Symbolic math, integration, differentiation, and generators (`sympy`)
- Analytics and data frames (`pandas`)

EXAMPLE: ROOT FINDING

- Find x_0 such that $f(x_0) = 0$.
- HW #1 – secant method
- Bisection algorithm
- Assumptions
  - Input, a and b float
  - Input f continuous with a single root in $[a, b]$, $f(a)f(b) < 0$
  - Input tol for termination tolerance (allowed error)
    - i.e., $|f(\hat{x}_0)| < tol$

BISECTION ALGORITHM

```
def (f, a, b, tol):
  x = (a+b)/2
  while abs(f(x)) > tol:
    if f(x)*f(a) > 0:
       a = x
    else:
      b = x
    x = (a+b)/2
  return x

f(cos, 1.5, 1.6, 10E-6)
```

BISECTION ALGORITHM USING SCIPY

- `scipy.optimize.bisect(f, a, b [, xtol])`
  - bisect until $|a - b| < xtol$, default $xtol = 10^{-12}$

```
from scipy.optimize import bisect

def f(x):
    return x ** 3 - 2 * x ** 2

x = bisect(f, a=1.5, b=3, xtol=1e-6)
print(f"Root found approx x={x:14.12g}")
# Root found approx x=2.000000023842
```

SCIPY FSOLVE ALGORITHM

```
from scipy.optimize import fsolve
# fsolve is multidimensional

def f(x):
   return x ** 3 - 2 * x ** 2

x = fsolve(f, x0=[1.6])
print(f"Root found approx x={x}")
# Root found approx x=[2.]
```

CALCULUS: DERIVATIVES

- Deep neural networks rely on robust derivative
  - But most of these have closed form and easy evaluation
- Need generic numeric methods for functions without analytic representations
- Two common methods that use a finite-difference technique
  - forward difference
  - backward difference

NUMERICAL DIFFERENTIATION

- Recall: $\frac{df}{dx}(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$

- Approximate with *forward difference* operator:

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}$$

Does not require knowledge of analytic derivative, only $f(x)$

Forward difference uses $f(x)$ and $f(x+h)$

Error: $E_{fw}(h) = -h\frac{f''(x)}{2!} - \frac{h^2 f'''(x)}{3!} + \cdots$

$$f'(x) = \frac{f(x+h)-f(x)}{} + \mathcal{O}(h)$$

IMPROVING NUMERICAL DIFFERENTIATION

- Consider *backward difference* operator:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Forward difference uses $f(x)$ and $f(x-h)$.

Similar behavior to forward.

Improve by (symmetric) combination, forward + backward (central)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

NUMPY

NUMPY

- numpy is an interface to high performance linear algebra libraries
  - ATLAS, LAPACK, BLAS

- Numpy's big contribution: the `array` object
  - very fast mathematical operations with arrays
  - linear algebra
  - fourier transforms
  - generating random numbers

- Numpy is an EXTERNAL module, not part of Python standard library

NUMPY ARRAYS ARE A SEQUENCE OF OBJECTS

- Think as vector - all objects in array must be same type

```
>>> import numpy as np
>>> a = np.array([1, 4, 16])
>>> type(a)
<class nump.ndarray>
>>> a.shape
(3,)
>>> np.sqrt(a)
array([1., 2., 4.])
>>> a > 3
array([False, True, True], dtype=bool)
```

Numpy - Datatypes

- a = np.array([1, 2, 3, 4])
  - a.dtype = int64
- a = a.astype(np.float32)
  - a = [1. 2. 3. 4.]
- a = a.astype(np.complex64)
  - a = [1.+0.j 2.+0.j 3.+0.j 4.+0.j]
- a = a.astype(bool)
  - a = [True True True True]
- a = a.astype(str)
  - a = ['True' 'True' 'True' 'True']

- a=np.random.rand(1000000).astype(np.float64)
  - Memory size of a = 8000000 bytes
- b = a.astype(np.float32)
  - Memory size of b = 4000000 bytes
- Precision loss
  - np.sum(a - b)
  - 8.81246785733314e-06

CREATE ARRAYS FROM ITERABLES

```
>>> from numpy import array
# 1Dim (vector)
>>> a = array([1, 4, 10])
>>> print(a)
[1, 4, 10]
# 2Dim (matrix)
>>> B = array([[0, 2.5], [11, 12]])
>>> B
array([[0., 2.5],
       [11., 12.]])
```

ARRAY SHAPE TUPLE: DIMENSION AND SIZE

```
>>> a.shape
(3, )    # 1d with 3 elements
>>> B.shape
(2, 2)    # 2d with 2 x 2 = 4 elements

# array.shape is mutable
>>> B.shape = (4, )
>>> B
array([0., 2.5, 11., 12.])
```

ARRAY SHAPE TUPLE: DIMENSION AND SIZE

- Array shape - array.shape
- Array size - array.size
- Array dimension - array.dimension

```
mu_vector = np.array([2, 3, 4])
sigma_vector = np.array([0.3, 0.5, 0.7])
num_samples = 1000
samples = np.random.normal(mu_vector, sigma_vector, (num_samples, 3))

# inpsect size, dimensions,
print("\n Multivariate normal samples shapes")
print(f"Sample shape: {samples.shape}")
print(f"Sample size: {samples.size}")
print(f"Sample dimensions: {samples.ndim}")

--------------------output--------------------------------
 Multivariate normal samples shapes
Sample shape: (1000, 3)
Sample size: 3000
Sample dimensions: 2
```

Numpy "Empty" dimensions

- Difference between an array of shape [5,1] and [5]
- array.T - transpose of the array

```
# create an array with empty dimensions
d1 = np.array([1,2,3,4,5])
d2 = np.array([[1,2,3,4,5]])
d3 = np.array([[1],[2],[3],[4],[5]])
print("\n Original arrays shapes")
print(f"d1 shape: {d1.shape}")
print(f"d2 shape: {d2.shape}")
print(f"d3 shape: {d3.shape}")

# Note: d2.T is the same as
print("\n Transpose of d2 is same as d3")
print(f"d2.T = \n {d2.T}")
print(f"d3 = \n {d3}")
```

```
——————————output———————————————————————
 Original arrays shapes
d1 shape: (5,)
d2 shape: (1, 5)
d3 shape: (5, 1)

 Transpose of d2 is same as d3
d2.T :[[1]
       [2]
       [3]
       [4]
       [5]]

d3 =  [[1]
       [2]
       [3]
       [4]
       [5]]
```

Numpy - add/remove “Empty” dimensions

- original array d1: [1 2 3 4 5]
- Add an empty dimension to 0st axis using **np.newaxis**
  d1[np.newaxis, :] : [[1 2 3 4 5]]
- Add an empty dimension to 1th axis using np.newaxis
  d1[:, np.newaxis] : [[1]
  [2]
  [3]
  [4]
  [5]]
- Add an empty dimension to 0th axis using **np.expand_dims**
  np.expand_dims(d1, axis=0) : [[1 2 3 4 5]]
- Add an empty dimension to 1th axis using np.expand_dims
  np.expand_dims(d1, axis=1) : [[1]
  [2]
  [3]
  [4]
  [5]]
- Remove an empty dimension using squeeze
- Original array d2: [[1 2 3 4 5]]
- **np.squeeze(d2)** : [1 2 3 4 5]
- Original array d3: [[1]
  [2]
  [3]
  [4]
  [5]]
- np.squeeze(d3) : [1 2 3 4 5]

ARRAY LENGTH AND MEMORY ALLOCATION

```
>>> a.size
3
>>> B.size
4

>>> a.nbytes
12
>>> B.nbytes
32
```

ARRAY ELEMENTS MUST ALL BE SAME TYPE

```
>>> a.dtype
dtype('int64')
>>> B.dtype
dtype('float64')

# type assignable on create
>>> a2 = array([1, 4, 10], numpy.float)
>>> a2.dtype
dtype('float64')
```

https://numpy.org/doc/stable/user/basics.types.html

CREATE FIXED SIZE ARRAY – FILL WITH 0 OR 1

```
>>> numpy.zeros((3, 3))   # tuple
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> numpy.zeros((4, ))     # same:
numpy.zeros(4)
array([0., 0., 0., 0.])
>>> numpy.ones((2, 4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

ARRAY INDEXING

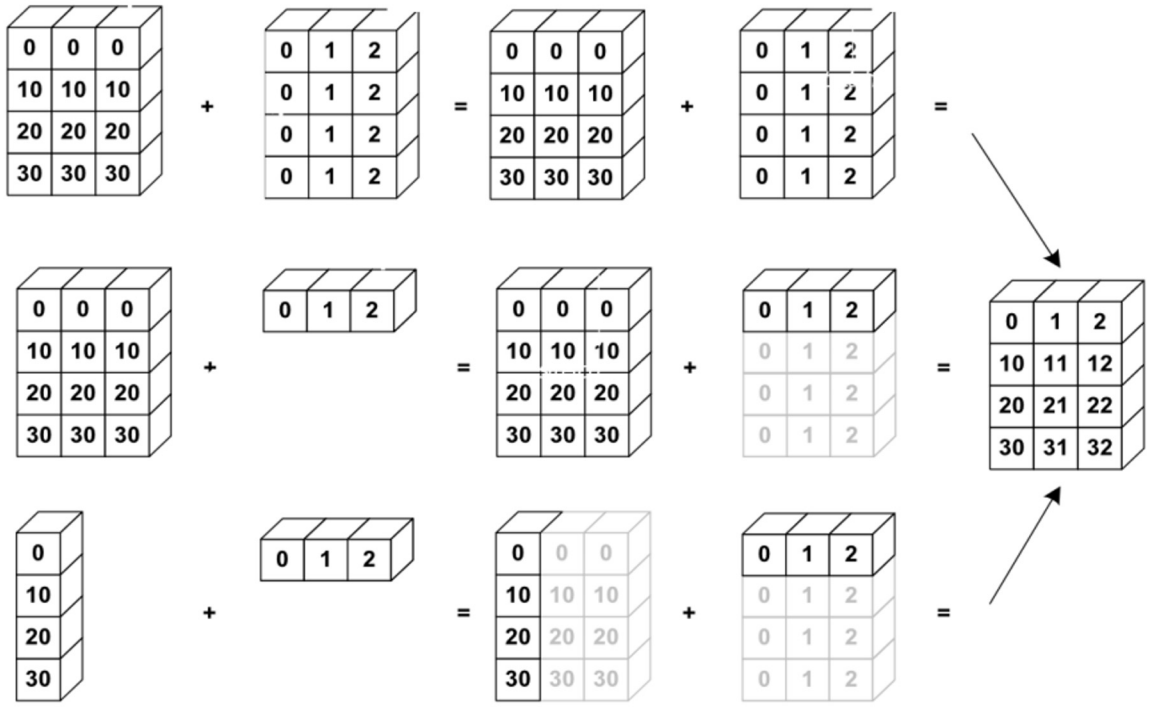
```
>>> x = numpy.array(range(0, 10, 2))        # length 5
>>> x[4]            # 8
>>> len(x)          # 5
>>> x.shape         # (5, )
>>> C = numpy.arrange(12)
>>> C.shape = (3,4)
array[[0, 1, 2, 3],
      [4, 5, 6, 7],
      [8, 9, 10, 11]]
>>> C[0, 0]         # 0
>>> C[2, -1]              # 11
>>> C[-1, -1]             # 11
```

ARRAY BROADCASTING

- NumPy compares shapes when operating on multiple shapes.

- Two dimensions are compatible if:

  1. Equal size, or
  2. One size = 1



Array Operations

a = [1 2 3 4]
b = [5 6 7 8]

- Element-wise addition/subtraction:
  - a + b = [6 8 10 12]
- Element-wise multiplication/division:
  - a * b = [5 12 21 32]
- Element-wise exponent
  - a ** b = [1   64 2187 65536]
- Scalar Operations
  - a + 1 = [2 3 4 5]

VECTOR OPERATIONS

- Inner product
- Outer product
- Dot product (matrix multiplication)

```
>>> u = [1, 2, 3]
>>> v = [1, 1, 1]
>>> np.inner(u, v)
6
>>> np.outer(u, v)
array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])
>>> np.dot(u,v)
6
```

NP.DOT

numpy.dot(*a, b, out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using **matmul** or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to **multiply** and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

https://numpy.org/doc/stable/reference/generated/numpy.dot.html

OPERATIONS OVER AXES

```
a = np.random.random((2,3))
# array([[ 0.9190687 , 0.36497813, 0.75644216],
#        [ 0.91938241, 0.08599547, 0.49544003]])
a.sum()
# 3.5413068994445549
a.sum(axis=0) # column sum (aggregate over rows)
# array([ 1.83845111, 0.4509736 , 1.25188219])
a.cumsum(axis=1) # cumulative row sum
# array([[0.9190687, 1.28404683, 2.04048899],
#        [0.91938241, 1.00537788, 1.50081791]])
a.min()
# 0.085995469040367 7
a.max(axis=0)
# array([ 0.91938241, 0.36497813, 0.75644216])
```

COPY AND RESHAPING ARRAY

```
>>> copy_x = x[:]              # or copy_x = x[::]
>>> rev_x = x[::-1]
>>> zero_x = numpy.zeros(x.shape)
```

- Reshape arrays with `np.reshape()`
  - "Total" size must remain the same (number of elements)
- Reshape arrays with `np.resize()`
  - Always works
  - Chops or appends zero
  - First dimension(s) have fill priority
  - **Can lead to unexpected results**

Numpy - Reshape/Resize

- original array d1: [1 2 3 4 5]

- Resize an array using **np.resize**
- np.resize(d1, (5, 1)) : [[1]
  [2]
  [3]
  [4]
  [5]]

- Reshape an array using **np.reshape**
- np.reshape(d1, (5, 1)) : [[1]
  [2]
  [3]
  [4]
  [5]]

- Original array d4: [[[1 2 0 7]
  [4 1 4 5]
  [3 1 8 1]]

  [[9 9 2 3]
  [0 2 9 7]
  [9 8 1 8]]]

- np.reshape(d4, (6, 4)) : [[1 2 0 7]
  [4 1 4 5]
  [3 1 8 1]
  [9 9 2 3]
  [0 2 9 7]
  [9 8 1 8]]

HIGHER DIMENSIONAL ARRAY SLICING

- Identical to 1-D but applied per component

```
>>> C
array([0, 1, 2, 3],
      [4, 5, 6, 7],
      [8, 9, 10, 11]])
>>> C[0, :]          # row 0
array([0, 1, 2, 3])
>>> C[:, 1]          # col 1
array([1, 5, 9])
```

USEFUL LINEAR ALGEBRA OPERATIONS

```
>>> import numpy.linalg
```

- `pinv`
  - matrix pseudo-inverse
- `svd`
  - singular value decomposition
- `det`
  - matrix determinant
- `eig`
  - eigenvalue and eigenvector pairs

Numpy - Matrix Operations

- a =
  [[1 2 3]
  [4 5 6]
  [7 8 9]]
- b =
  [[1 2 4]
  [4 2 6]
  [1 8 9]]

- Matrix multiplication
  - a @ b =
    [[12 30 43]
    [30 66 100]
    [48 102 157]]

- Note this is different from element-wise multiplication
  - a * b =
    [[1 4 12]
    [16 10 36]
    [7 64 81]]

- Matrix transpose
  - a.T =
    [[1 4 7]
    [2 5 8]
    [3 6 9]]

- Matrix inverse (if matrix is invertible)
  - a is not invertible
  - b^-1 = **np.linalg.inv**(b) =
    [[-1. 0.46666667 0.13333333]
    [-1. 0.16666667 0.33333333]
    [1. -0.2 -0.2]]

- Matrix determinant
  - det(a) = **np.linalg.det**(a) = 0.0
  - det(b) = np.linalg.det(b) = 30.0

SOLVING A LINEAR SYSTEM OF EQUATIONS

- `numpy.linalg.solve(A, b)` solves $Ax = b$ for square matrix A and vector b. Return solution vector x as an array.

```
>>> A = numpy.array([[1, 0], [0, 2]])
>>> b = numpy.array([1, 4])
>>> from numpy import linalg AS LA
>>> x = LA.solve(A, b)
array([1., 2.])
>>> numpy.dot(A, x) # verify = b
array([1., 4.])
```

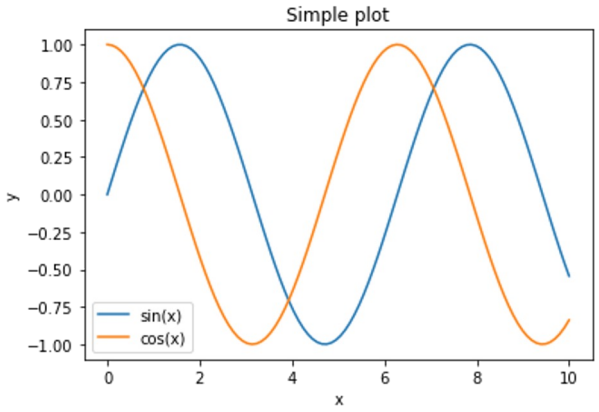
MATPLOTLIB

PLOTTING WITH MATPLOTLIB

- Matplotlib is the de facto plotting library
  - keep easy tasks easy, and hard tasks possible
- Full 2D+3D plotting library
  - publication quality figures
- Can run fully scripted
  - or Interactive
- Jupyter/IPython configurations
  - `%matplotlib inline`
  - `%matplotlib qt`
  - `%matplotlib notebook`

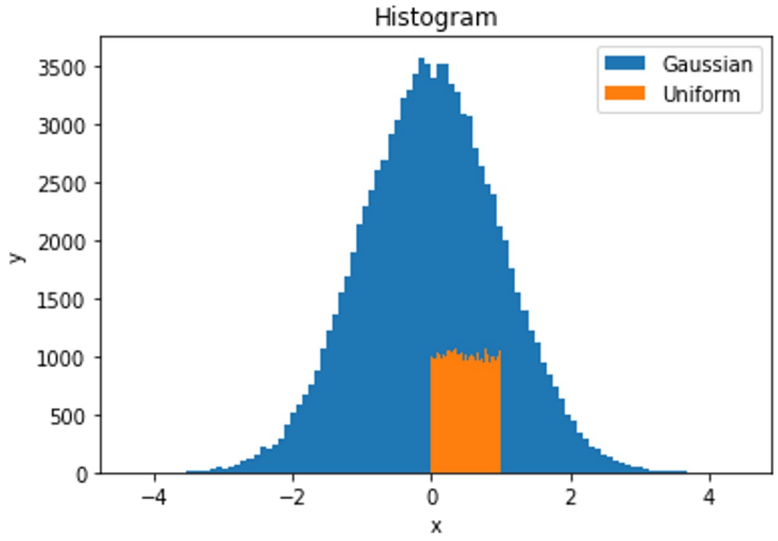
Line Plot

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x, y, label="sin(x)")
plt.plot(x, z, label="cos(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Simple plot")
plt.legend()
plt.show()
```

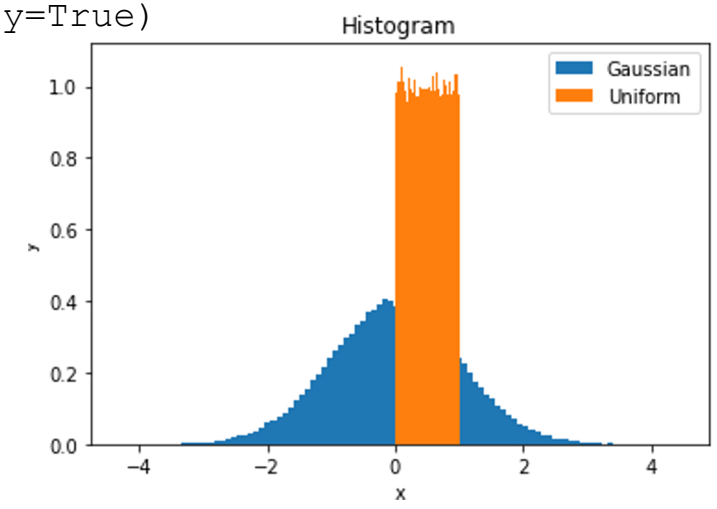


Histogram

```
x = np.random.randn(100000)
y = np.random.uniform(0, 1, 100000)
plt.hist(x, bins=100, label="Gaussian")
plt.hist(y, bins=100, label="Uniform")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Histogram")
plt.legend()
plt.show()
```

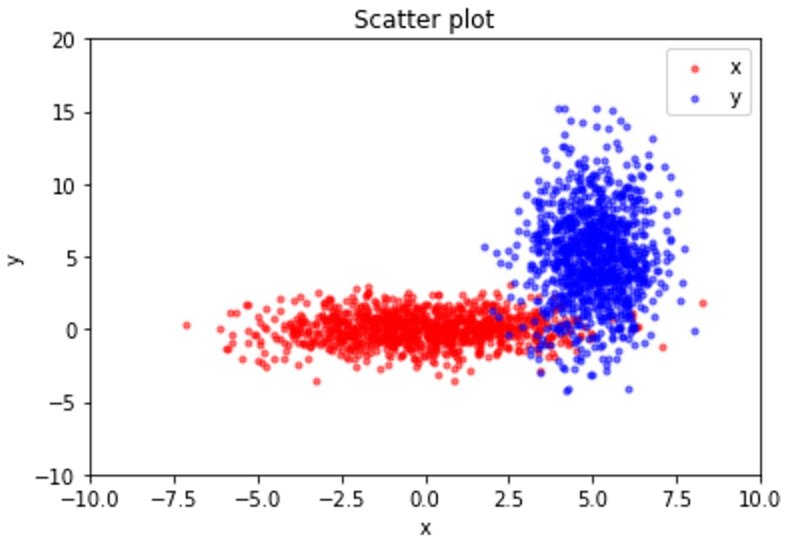



```
plt.hist(x, bins=100, label="Gaussian", density=True)
plt.hist(y, bins=100, label="Uniform", density=True)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Histogram")
plt.legend()
plt.show()
```



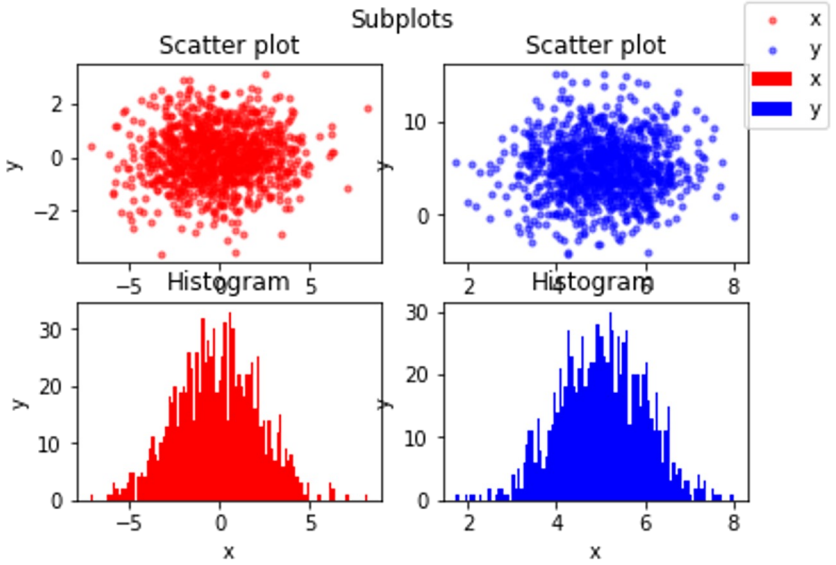
Scatter plots

```
# scatter plots
# generate two 2D gaussian samples
x = np.random.multivariate_normal([0, 0], [[5, 0], [0, 1]], 1000)
y = np.random.multivariate_normal([5, 5], [[1, 0], [0, 10]], 1000)

# plot the samples
plt.scatter(x[:, 0], x[:, 1], label="x", color="red", alpha=0.5, s=10)
plt.scatter(y[:, 0], y[:, 1], label="y", color="blue", alpha=0.5, s=10)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Scatter plot")
# set axis limits
plt.xlim(-10, 10)
plt.ylim(-10, 20)

plt.legend()
plt.show()
```



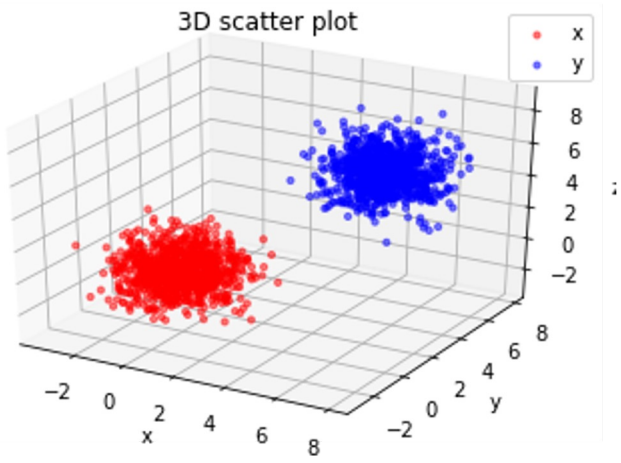
Subplots

```
fig, ax = plt.subplots(2, 2)
ax[0, 0].scatter(x[:, 0], x[:, 1], label="x", color="red", alpha=0.5, s=10)
ax[0, 1].scatter(y[:, 0], y[:, 1], label="y", color="blue", alpha=0.5, s=10)
ax[1, 0].hist(x[:, 0], bins=100, label="x", color="red")
ax[1, 1].hist(y[:, 0], bins=100, label="y", color="blue")
# set labels for all subplots
for i in range(2):
    for j in range(2):
        ax[i, j].set_xlabel("x")
        ax[i, j].set_ylabel("y")
# set titles for each subplot
ax[0, 0].set_title("Scatter plot")
ax[0, 1].set_title("Scatter plot")
ax[1, 0].set_title("Histogram")
ax[1, 1].set_title("Histogram")
# set title for the whole figure
fig.suptitle("Subplots")
# set legend for the whole figure
fig.legend()
# increase space between upper and lower subplots
fig.subplots_adjust(hspace=0.5)
plt.show()
```



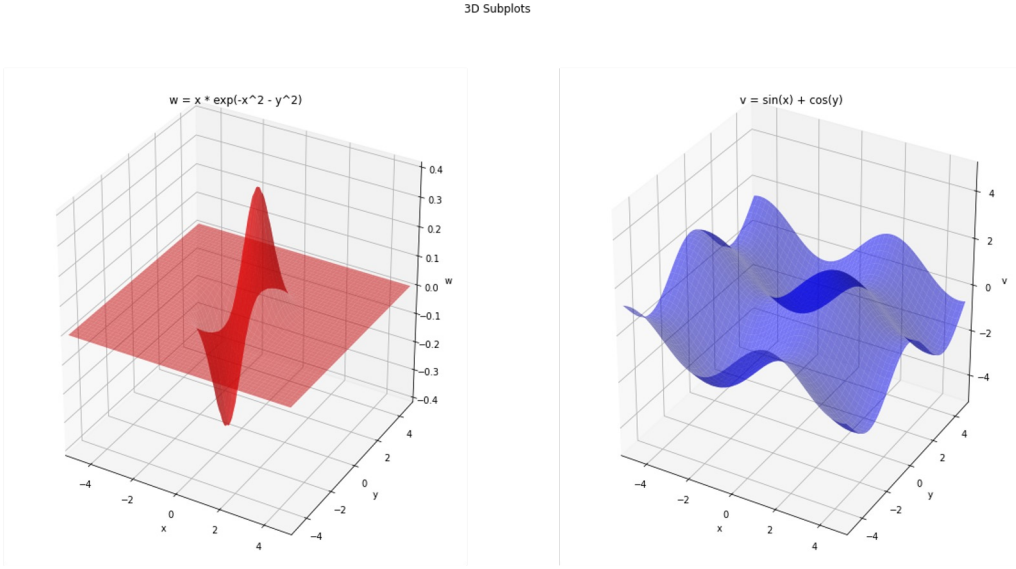
3D scatter

```
# create a simple 3d figure
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

# generate 3D gaussian samples
x = np.random.multivariate_normal([0, 0, 0], [[1, 0, 0], [0, 1, 0], [0, 0, 1]], 1000)
y = np.random.multivariate_normal([5, 5, 5], [[1, 0, 0], [0, 1, 0], [0, 0, 1]], 1000)

# plot the samples
ax.scatter(x[:, 0], x[:, 1], x[:, 2], label="x", color="red", alpha=0.5, s=10)
ax.scatter(y[:, 0], y[:, 1], y[:, 2], label="y", color="blue", alpha=0.5, s=10)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")
ax.set_title("3D scatter plot")
ax.legend()
plt.show()
```



3D Mesh

```
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)
v = np.sin(x) + np.cos(y)
w = x*np.exp(-x**2 - y**2)
fig = plt.figure(figsize=(20, 10))
ax1 = fig.add_subplot(121, projection="3d")
ax1.plot_surface(x, y, w, label="w", color="red", alpha=0.5)
ax2 = fig.add_subplot(122, projection="3d")
ax2.plot_surface(x, y, v, label="v", color="blue", alpha=0.5)
ax1.set_xlabel("x")
ax1.set_ylabel("y")
ax1.set_zlabel("w")
ax1.set_title("w = x * exp(-x^2 - y^2)")
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.set_zlabel("v")
ax2.set_title("v = sin(x) + cos(y)")
ax1.set_xlim(-5, 5)
ax1.set_ylim(-5, 5)
ax1.set_zlim(-0.4, 0.4)
ax2.set_xlim(-5, 5)
ax2.set_ylim(-5, 5)
ax2.set_zlim(-5, 5)
fig.suptitle("3D Subplots")
```



MATPLOTLIB.PYPLOT

- `Matplotlib.pyplot` is object-oriented plotting interface
  - fine grained control over all aspects of plotting

- Impressive demo gallery

https://matplotlib.org/2.0.2/gallery.html

- Instructive demo notebook

https://www.desy.de/~fangohr/teaching/csds2019/notebooks/Matplotlib.html

FUNCTIONAL PROGRAMMING

FUNCTIONAL TOOLS

- List processing is a common pattern
- `for`-loops are conceptually simple
  - not compact and not always performant
- Other options:
  - list comprehension
  - `map`, `filter`, and `reduce` (lambda functions)

LAMBDA FUNCTIONS ARE ANONYMOUS FUNCTIONS

Useful to define small *helper* functions - usually single purpose use

```
>>> lambda a: a
<function <lambda> at 0x319c60>
>>> lambda a: 2 * a
<function <lambda> at 0x319ce0>
>>> (lambda a: 2 * a)(10)
20
>>> (lambda x, y, z: (x + y) * z)(10, 20, 2)
60
>>> type(lambda x, y: x + y)
<type 'function'>
```

LAMBDA VS DECLARED FUNCTION

```
from scipy.integrate import quad
def f(x):
  return x * x   # x^2
y, abserr = quad(f, a=0, b=2)
print(f"Int f(x)=x^2 from 0 to 2 = {y} +/-
{abserr}")
```

```
from scipy.integrate import quad
y, abserr = quad(lambda x: x * x, a=0, b=2)
print(f"Int f(x)=x^2 from 0 to 2 = {y} +/-
{abserr}")
```

HIGHER ORDER FUNCTIONS

- functions that take or return functions

- `map(function, iterable) → iterable`
  - apply function each element in iterable, returns a new iterable

- `filter(function, iterable) → iterable`
  - return new iterable containing only elements with function(item) is true

- `reduce(function, iterable, initial) → value`
  - apply function(x, y) from left to right - reduces iterable to single value

MAPPING ITERABLES

```
>>> def f(x): return x ** 2
>>> map(f, [0, 1, 2, 3, 4])
<map object at 0x1026a52e0>
>>> list(map(f, [0, 1, 2, 3, 4]))
                [0, 1, 4, 9, 16]
>>> list(map(lambda x: x ** x, [0, 1, 2, 3, 4]))
>>> [x ** 2 for x in [0, 1, 2, 3, 4]]

>>> import math
>>> list(map(math.exp, [0, 0.1, 1.]))
```

FILTERING ITERABLES

```
>>> c = "The quick brown fox jumps".split()
['The', 'quick', 'brown', 'fox', 'jumps']
>>> list(map(lambda s: len(s) > 4, c))
[False, True, True, False, True]
>>> list(filter(lambda s: len(s) > 4, c))
['quick', 'brown', 'jumps']

# compare
>>> [s for s in c if len(s) > 4]
```

REDUCE AN ITERABLE TO A VALUE

```
>>> from functools import reduce
>>> def f(x, y):
…     print(f"Called with x={x}, y={y}")
…     return x + y
>>> reduce(f, [1, 3, 5], 0)
9
>>> reduce(f, [1, 3, 5], 100)
109
```

FUNCTIONAL PROGRAMMING

- map, reduce, and filter are common across language
  - allows consistent patterns to deal with lists and iterables
- they functionally abstract common loop tasks
- Recommendation: use these instead of loops (or list comprehension)
  - loops are tedious (and error prone)
  - these functional equivalents often clearer to read
  - resulting code *can* be faster (lambda functions can be slower)
  - if you need to use lambda. often better off with list comprehension

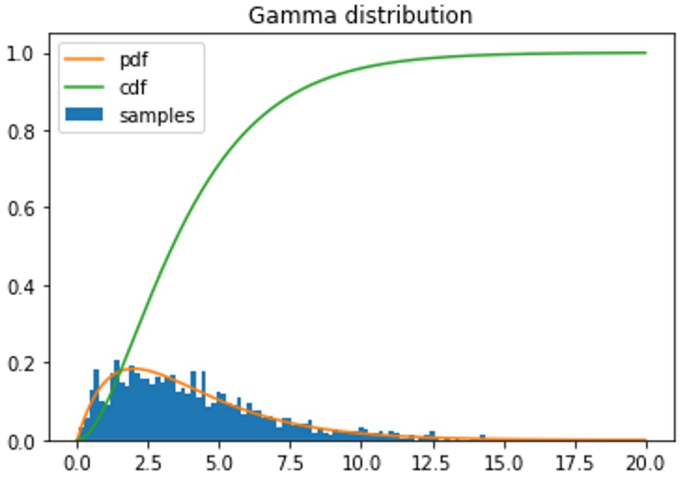
SCIPY

SCIENTIFIC PYTHON (SCIPY)

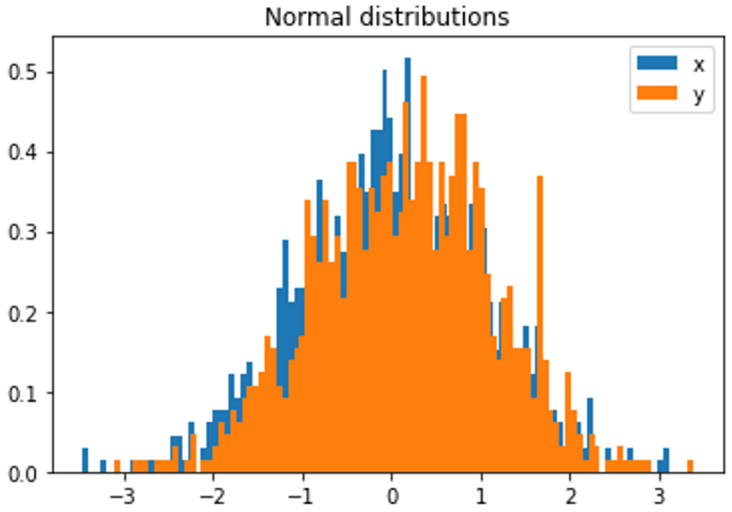
- `stats` — Statistical functions
- `sparse` — Sparse matrices
- `linalg` — Linear algebra
- `signal` — Signal processing
- `interpolate` — Interpolation
- `optimize` — Optimization
- `cluster` — Vector quantization / K-means
- `fftpack` — Discrete Fourier Transforms
- `Integrate` — Integration

SCIPY - stats

```
# with stats we have access to a lot of statistical distributions, their pdfs, cdfs, their moments, tests, etc.
x = stats.gamma.rvs(2, scale=2, size=1000)
plt.hist(x, bins=100, density=True, label="samples")
x = np.linspace(0, 20, 1000)
plt.plot(x, stats.gamma.pdf(x, 2, scale=2), label="pdf")
plt.plot(x, stats.gamma.cdf(x, 2, scale=2), label="cdf")
```




```
# statistical tests
x = stats.norm.rvs(0, 1, size=1000)
y = stats.norm.rvs(0.15, 1, size=1000)
plt.hist(x, bins=100, density=True, label="x")
plt.hist(y, bins=100, density=True, label="y")
print(stats.ttest_ind(x, y))

#Ttest_indResult(statistic=-3.357944448803533,
pvalue=0.0007999795533366586)
```



SCIPY - sparse

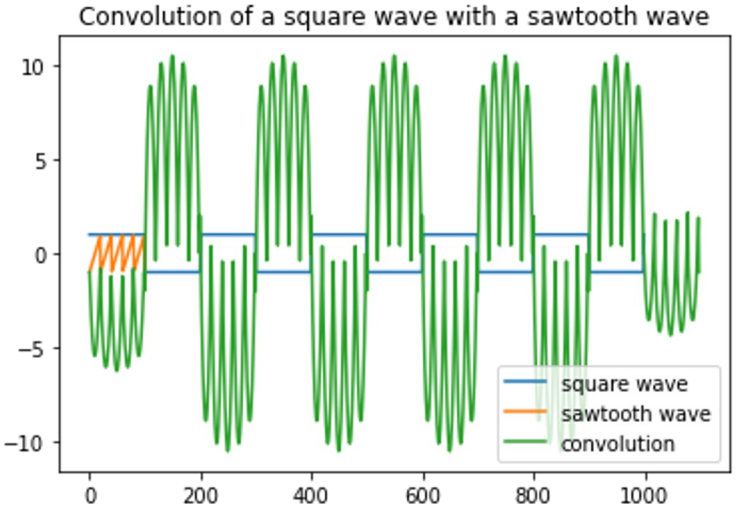
```
# with sparse we can work with sparse matrices, to save memory and speed up computations
# example: create a 1000x1000 matrix with 1s on the diagonal and 0s everywhere else
x = sparse.eye(1000, 1000)

# create corresponding dense matrix
x_dense = x.todense()

# print the memory usage of the sparse and dense matrices
print(x.data.nbytes, x_dense.nbytes)
# 8000 8000000

# perform matrix multiplication
%timeit x_dense.dot(x_dense)
%timeit x.dot(x)
# 82.4 ms ± 23.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
# 507 µs ± 23.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

SCIPY - linalg

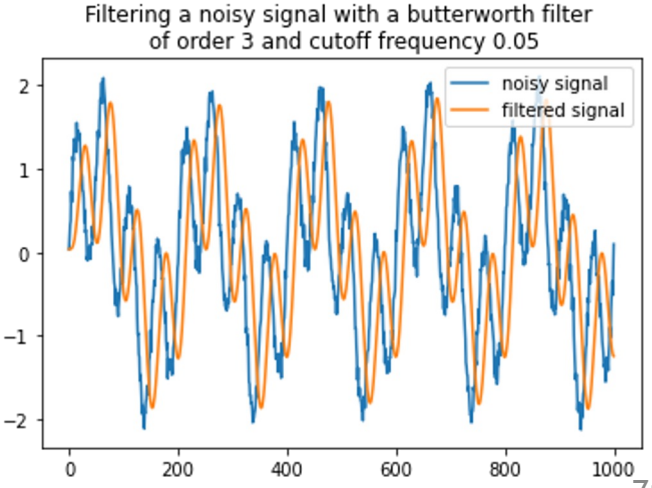
```
from scipy import linalg
# with linalg we can perform linear algebra operations like matrix
multiplication, solving linear systems, singular value
decomposition, get eigenvalues and eigenvectors, etc.

# SVD
x = np.array([[1, 4, 2], [6, 3, 0], [5, 2, 1], [2, 1, 3], [4, 5,
2]])
u, s, vh = linalg.svd(x)


# solve a linear system
# create a 3x3 matrix with random values, non singular
A = np.array([[1, 2, 3], [6, 3, 0], [5, 2, 1]])
# create a 3x1 vector with random values
b = np.array([1, 2, 3])
x = linalg.solve(A, b)


# get the eigenvalues and eigenvectors of a matrix
A = np.array([[1, 2, 3], [6, 3, 0], [5, 2, 1]])
eigenvalues, eigenvectors = linalg.eig(A)
```

SCIPY - signal

```
# signal processing from scipy
from scipy import signal
# with signal we can perform signal processing operations
like filtering, convolution, etc.

# create a 1000x1 vector with square wave values
x = signal.square(2 * np.pi * 5 * np.linspace(0, 1, 1000))
# create a 100x1 vector with a sawtooth wave
h = signal.sawtooth(2 * np.pi * 5 * np.linspace(0, 1, 100))
# perform convolution
y = signal.convolve(x, h)
```



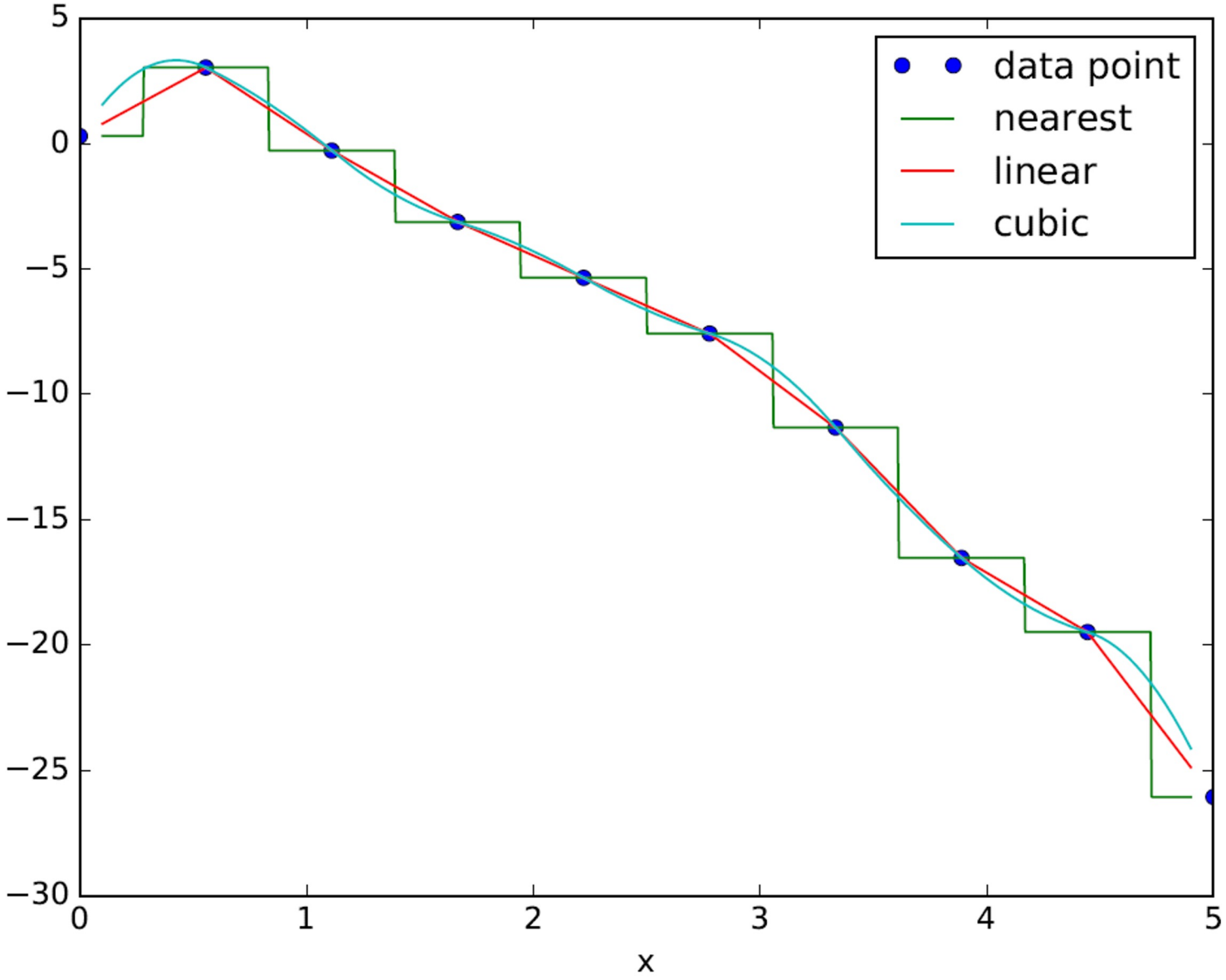

```
# filter a signal
# create a 1000x1 vector with sine wave values with frequency
component 5 and 20
base = np.linspace(0, 1, 1000)
x = np.sin(2 * np.pi * 5 * base) + np.sin(2 * np.pi * 20 *
base)
# add some noise
x += np.random.randn(len(x)) * 0.1
# create a butterworth filter
b, a = signal.butter(3, 0.05)
# apply the filter
zi = signal.lfilter_zi(b, a)
z, _ = signal.lfilter(b, a, x, zi=zi*x[0])
```



SCIPY - INTERPOLATION EXAMPLE

```
import numpy as np
import scipy.interpolate
import pylab

def create_data(n):
    """Given integer n return n data points x and y values"""
    xmax = 5.
    x = np.linspace(0, xmax, n)
    y = -x ** 2
    y += 1.5 * np.random.normal(size=len(x))      # additive
Gaussian noise
    return x, y

[…]
```

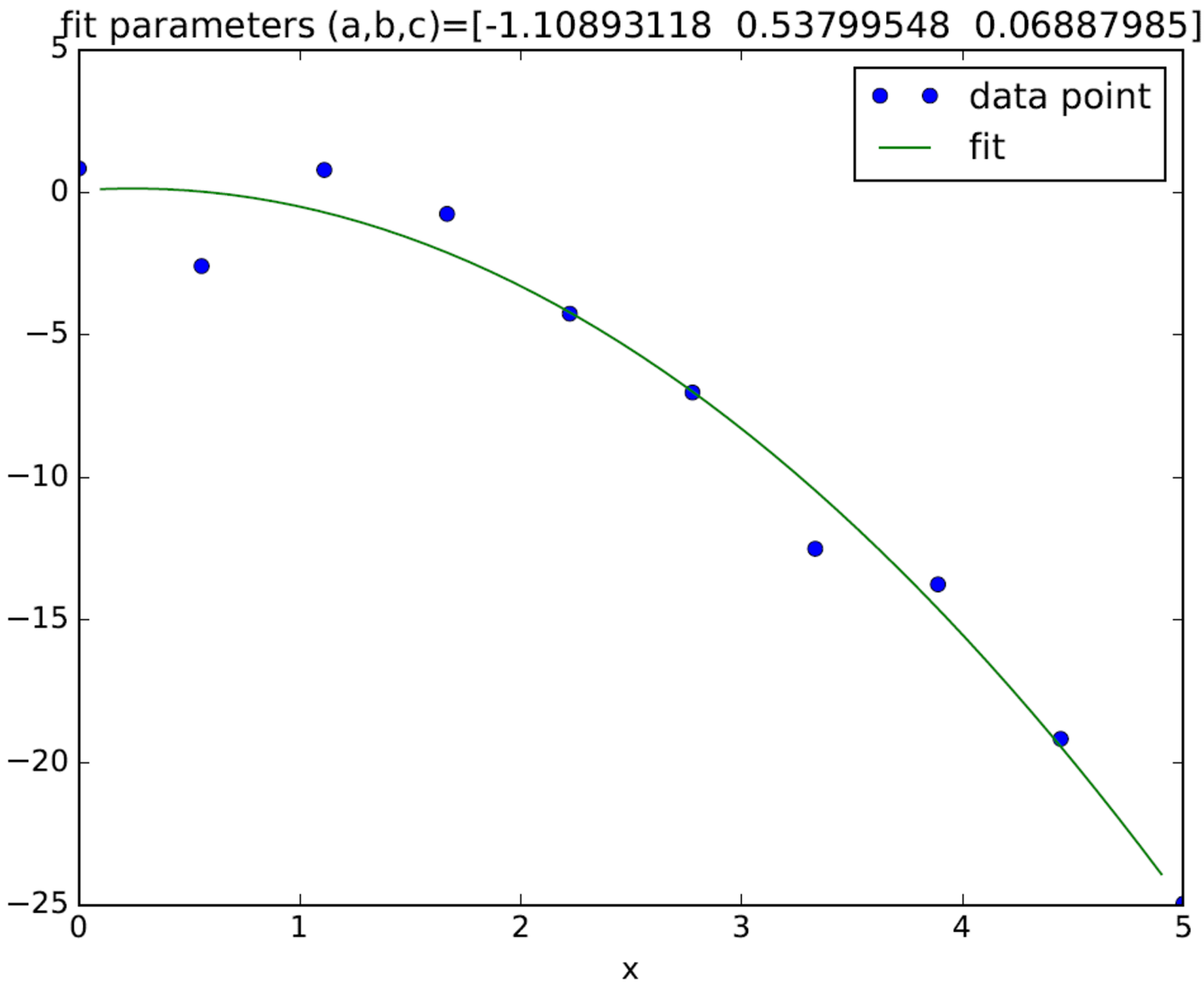
```
n = 10
x, y = create_data(n)
# plot mesh
xf = np.linspace(0.1, 4.9, n * 100)
# piecewise const, linear, quadratic
y0 = scipy.interpolate.interp1d(x, y, kind="nearest")
y1 = scipy.interpolate.interp1d(x, y, kind="linear")
y2 = scipy.interpolate.interp1d(x, y, kind="quadratic")

pylab.plot(x, y, 'o', label='data')
pylab.plot(xf, y0(xf), label='nearest')
pylab.plot(xf, y1(xf), label='linear')
pylab.plot(xf, y2(xf), label='quadratic')

pylab.legend()
pylab.label('x')
pylab.show()
```


data point
nearest
linear
cubic
5
0
−5
−10
−15
−20
−25
−30
0
1
2
3
4
5
x

SCIPY - optimize CURVE FITTING EXAMPLE

```
import numpy as np
import scipy.optimize
import pylab

def create_data(n):
    """Given integer n return n data points x and y values"""
    xmax = 5.
    x = np.linspace(0, xmax, n)
    y = -x ** 2
    y += 1.5 * np.random.normal(size=len(x))     # additive Gaussian noise
    return x, y

def model(x, a, b, c):
     return a * x ** 2 + b * x + c
```

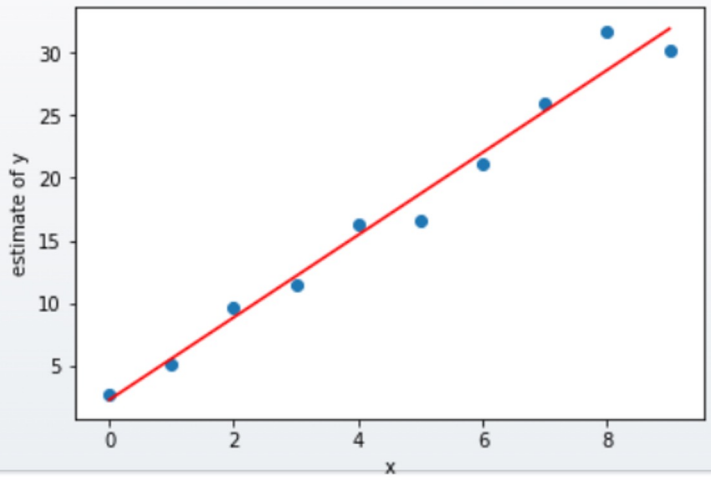
```
n = 10
x, y = create_data(n)
# fit
p, pcov = scipy.optimize.curve_fit(model, x, y)
a, b, c = p
# plot mesh
xf = np.linspace(0.1, 4.9, n * 100)

pylab.plot(x, y, 'o', label='data')
pylab.plot(xf, model(xf, a, b, c), label='fit')

pylab.title(f"fit parameters (a,b,c)=({a}, {b}, {c})")
pylab.legend()
pylab.label('x')
pylab.show()
```


fit parameters (a,b,c)=[-1.10893118 0.53799548 0.06887985]
data point
fit
5
0
−5
−10
−15
−20
−25
0
1
2
3
4
5
x

REGRESSION (CURVE-FITTING) FROM DATA

- Assume: data generated with known model + noise
- Guess: at model in practice (*e.g.*, linear)
- Choose: parameters to minimize error

```
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(10)
y = 3*x+4
y =  y + np.random.normal(0,2,10)
slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)
y_hat = intercept + slope * x

fig = plt.figure()
plt.plot(x,y_hat, color='r')
plt.scatter(x,y)
plt.xlabel("x")
plt.ylabel("estimate of y")
#axes = plt.gca()
#axes.set_xlim([-1, 4])
```

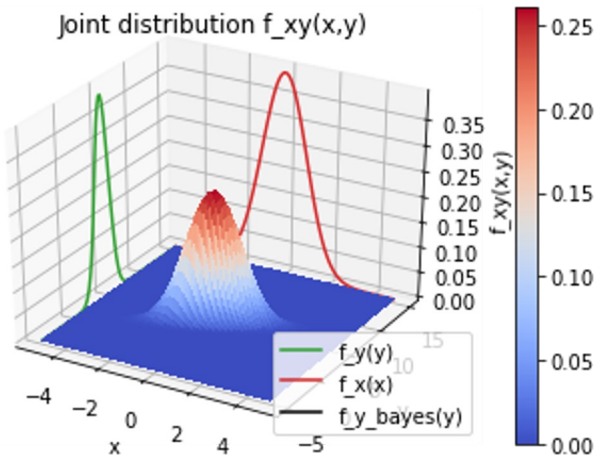


DEFINITE INTEGRAL EXAMPLE

```
from math import cos, exp, pi
from scipy.integrate import quad

def f(x):
    return exp(cos(-2 * x * pi)) + 3.2

# integrate [-2, 2]
res, err = quad(f, -2, 2)

print(f"Result is: {res} +/- {err}")
# The numerical result is 17.864264 (+-1.55117e-11)
```

DEFINITE INTEGRAL EXAMPLE (Bayesian partition integral)

```
from scipy import integrate
import scipy

mu = np.array([0, 4])
sigma = np.array([[1, 0.8], [0.8, 1]])

f_yx = lambda x,y: scipy.stats.multivariate_normal.pdf([x,y], mean=mu, cov=sigma)
f_x = lambda x: integrate.quad(lambda y: f_xy(x, y), -np.inf, np.inf)[0]
f_y = lambda y: integrate.quad(lambda x: f_xy(x, y), -np.inf, np.inf)[0]

def conditional_distribution(x, f_xy, f_x):
    return lambda y: f_xy(x, y)/ f_x(x)
```




```
f_y_given_x = lambda x: conditional_distribution(x, f_xy, f_x)
f_y_bayes = lambda y: integrate.quad(lambda x: f_y_given_x(x)(y)*f_x(x), -np.inf, np.inf)[0]
```

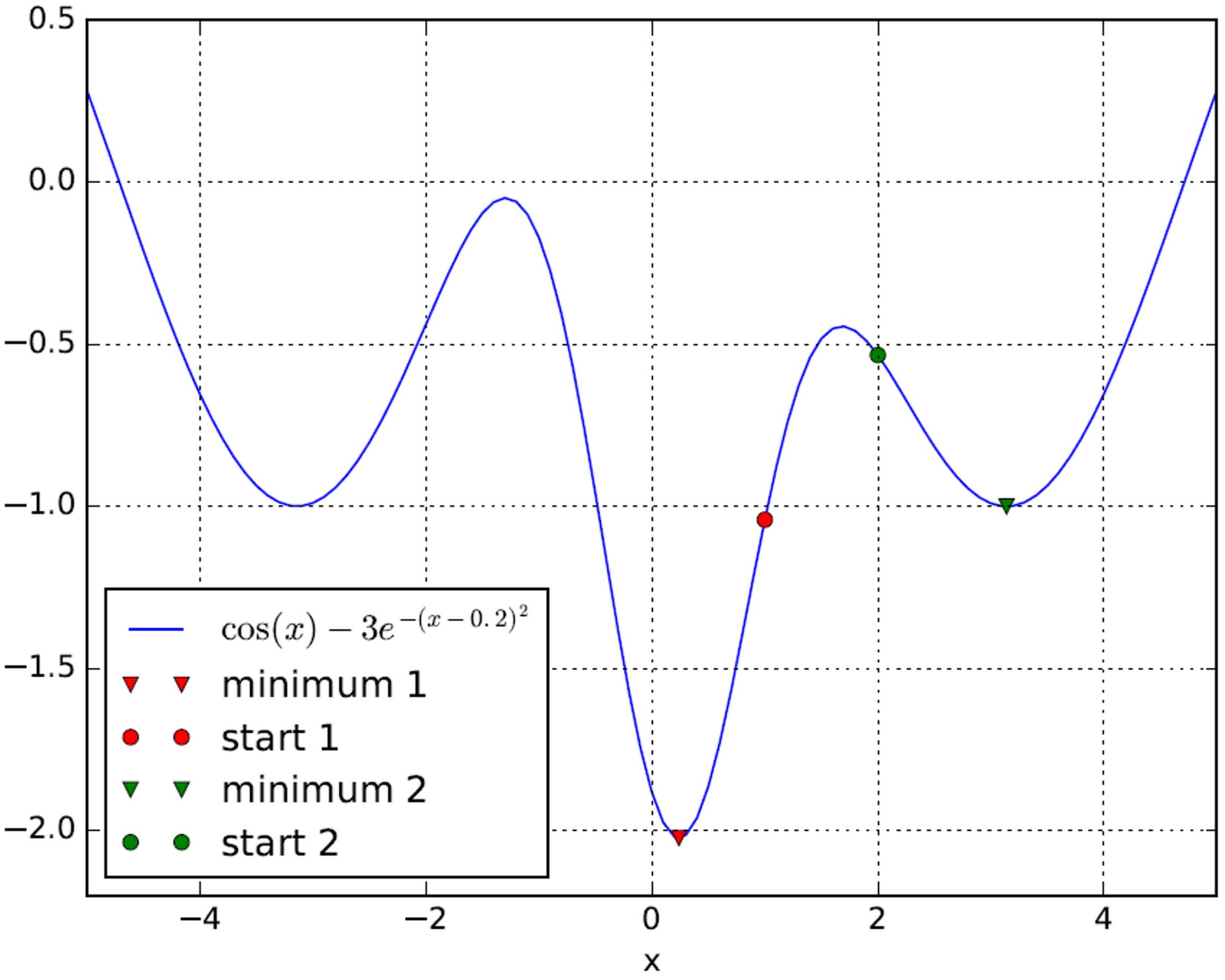
EE 541: OPTIMIZATION / MINIMIZATION

- Given a function $f(x)$ find x_m so that $f(x_m) < f(x)$ for $\forall x$.
  - Usually have to accept a local minimum (i.e., not global minimum)
  - $f(x_m) < f(x)$ for $\forall x: |x - x_m| < \epsilon$
- Maximization of $f(x)$ is minimization of $-f(x)$
  - does not require separate routines
- Most algorithms require a starting point(s) x_0 - "guesses"
  - "better" guesses usually closer to x_m

OPTIMIZATION EXAMPLE

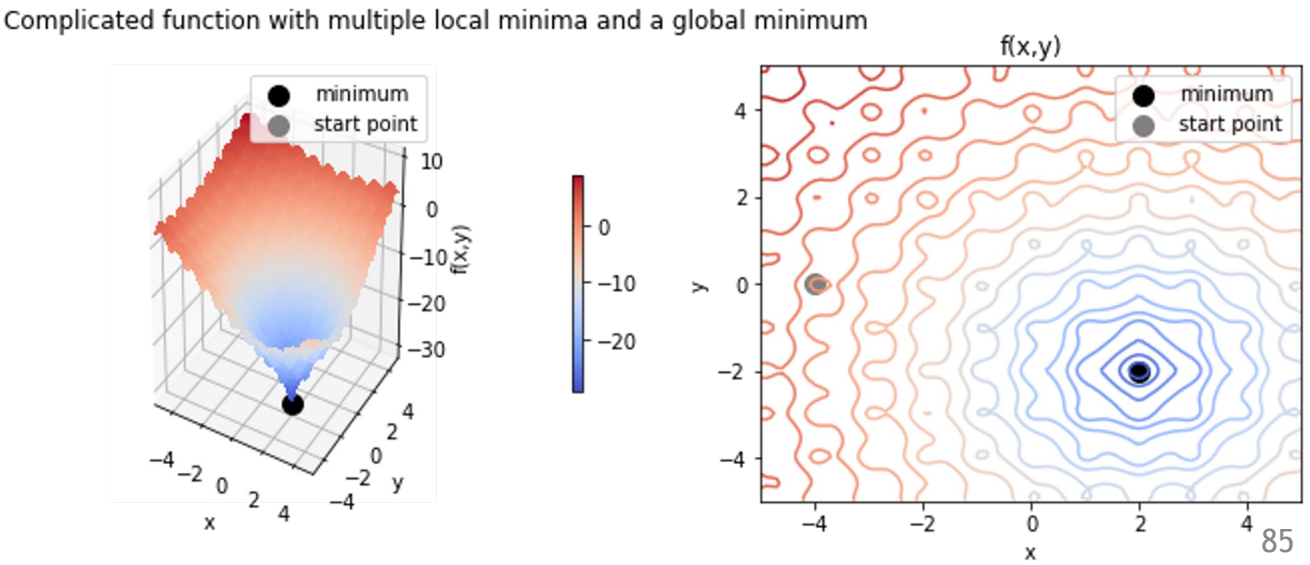
```
from scipy import arrange, cos, exp
from scipy.optimize import fmin
import pylab

def f(x):
    return cos(x) - 3 * exp( -(x - 0.2) ** 2
)
# find minima using 2 different x0
min1 = fmin(f, 1.0)          # [0.23964844]
min2 = fmin(f, 2.0)          # [3.13847656]
```

```
x = arrange(-10, 10, 0.1)
y = f(x)
pylab.plot(x, y, label, '$\cos(x) - 3 e^{-(x-
0.2)^2}$')
pylab.xlabel('x')
pylab.grid()
pylab.axis([-5, 5, -2.2, 0.5])
# add initial guess and min to plot
pylab.plot(min1, f(min1), 'vr', label='minimum 1')
pylab.plot(1.0, f(1.0), 'or', label='start 1')
pylab.plot(min2, f(min2), 'vg', label='minimum 2')
pylab.plot(2.0, f(2.0), 'og', label='start 2')
pylab.legend(loc='lower left')
pylab.show()
```


0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−2
0
2
4
x
$\cos(x) - 3e^{-(x-0.2)^2}$
minimum 1
start 1
minimum 2
start 2

2D Optimization

```
def f(x, y, x_0, y_0): #x_0, y_0 is the mean (and minimum) here
                return -50.0 * np.exp(-0.2 * np.sqrt(0.5 * ((x-x_0)**2 + (y-y_0)**2))) - \
                np.exp(0.5 * (np.cos(2 * np.pi * (x-x_0)) + np.cos(2 * np.pi * (y-y_0)))) + \
                np.e + \
                20
x0 = np.array([-4, 0]) # starting point
res = scipy.optimize.minimize(lambda x: f(x[0], x[1], x_0, y_0), x0, method='BFGS') #res.x - solution
```



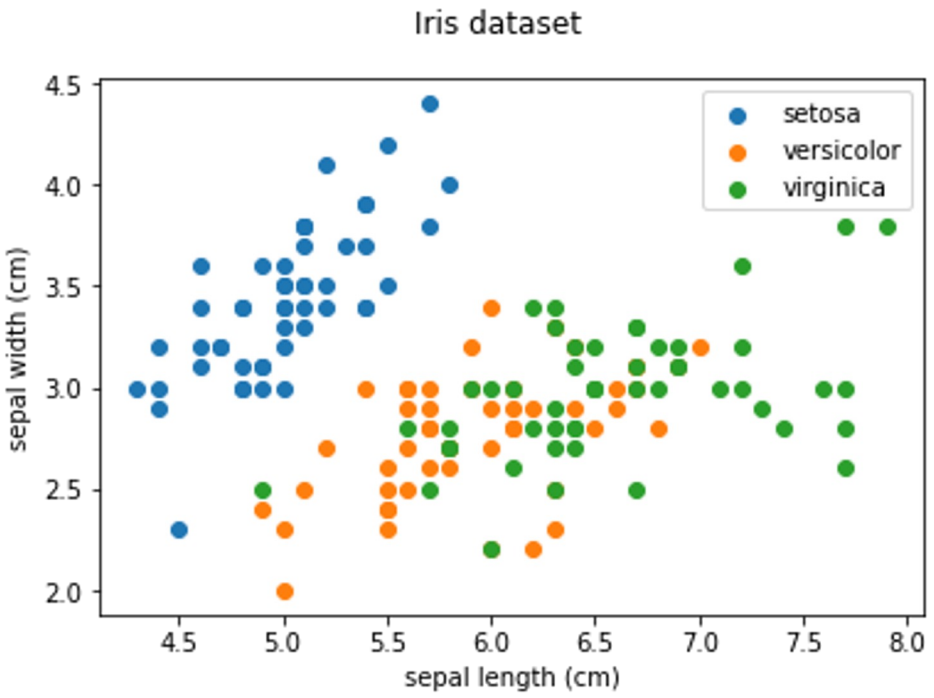
SKLEARN

SciKit - Learn(`sklearn`)

- `datasets` Few lightweight benchmarking datasets
- `clustering` Clustering models like KMeans
- `linear_model` Linear Regression models
- `decomposition` PCA
- `model_selection` Grid search and test_train_split
- `neural_network` MLP models

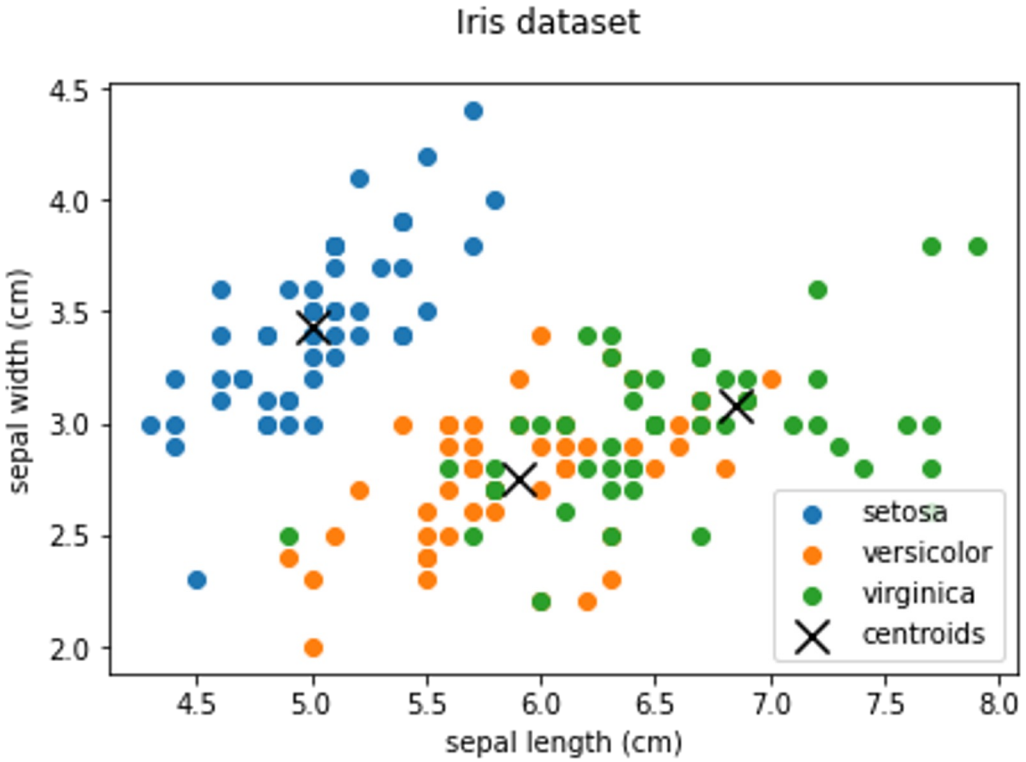
Sklearn - Datasets

```
# Eg, Iris Dataset:
# 3 classes of 50 instances each,
# each class refers to a type of iris plant
# 4 features: sepal length, sepal width, petal length and petal width

iris = sklearn.datasets.load_iris()
X = iris.data
y = iris.target
```



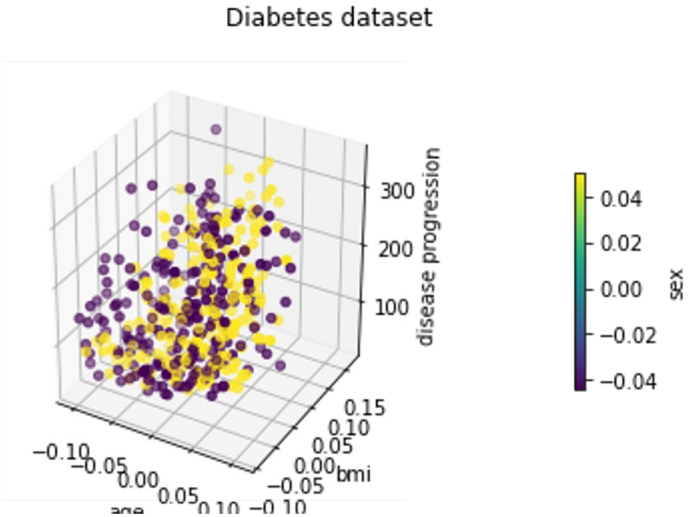
Sklearn - cluster

```
#clustering the iris dataset
kmeans = sklearn.KMeans(n_clusters=3)
kmeans.fit(X)
C = kmeans.cluster_centers_
```



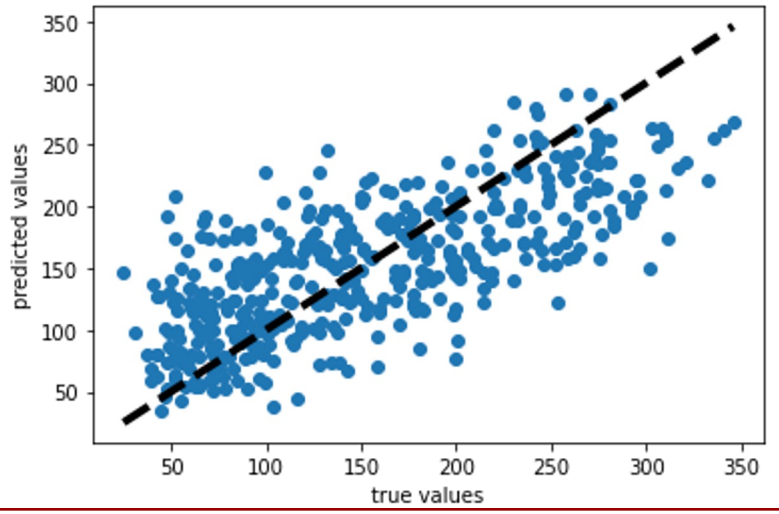
Sklearn - linear_model

```
# train a linear regression model on the diabetes dataset
diabetes = datasets.load_diabetes()
X_orig = diabetes.data
y_orig = diabetes.target
# plot first 3 features vs target
```



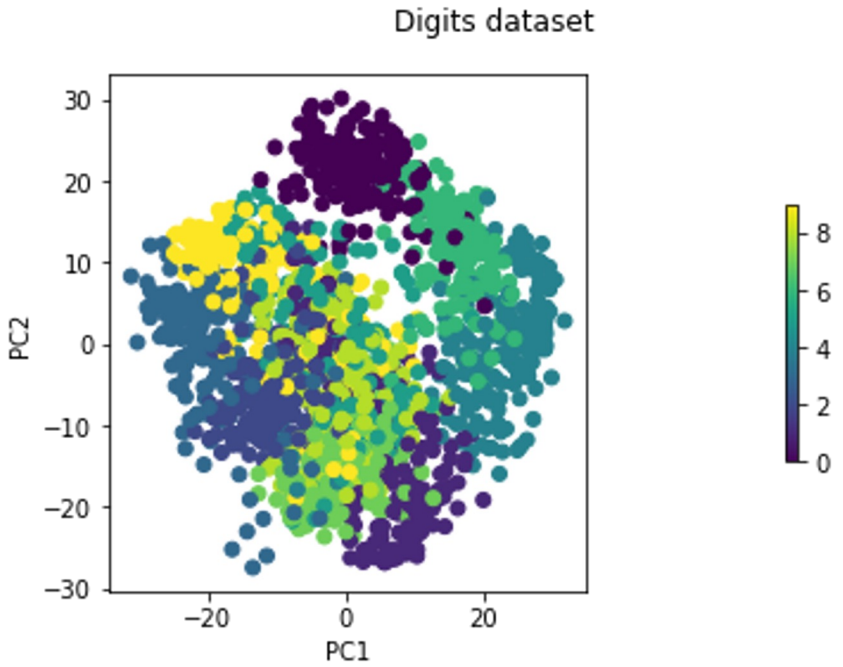

```
regr = sklearn.linear_model.LinearRegression()
regr.fit(X_orig, y_orig)
y_pred = regr.predict(X_orig)
```

```
# plot predicted target values vs true target values
# should ideally be on the straight line
```



Sklearn - decomposition

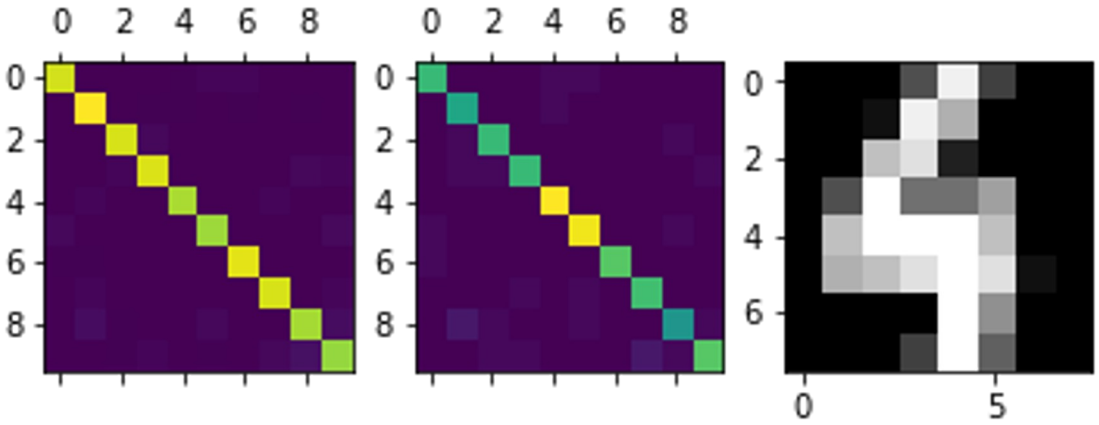
```
# PCA on the digits dataset
# 10 classes of 1797 instances
# Each class is a digit
# 64 features: 8x8 pixel values

digits = datasets.load_digits()
X = digits.data
y = digits.target

pca = sklearn.decomposition.PCA(n_components=2)
pca.fit(X)
X_pca = pca.transform(X)
```



Sklearn - neural_network

```
# on the digit dataset, we train an MLP classifier

clf = sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(8, 8), solver='adam',
learning_rate_init=0.0001, alpha=0.0001, max_iter=1000, activation='relu')

# split dataset into train and test
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X, y,
test_size=0.2, random_state=42)
clf.fit(X_train, y_train)
```

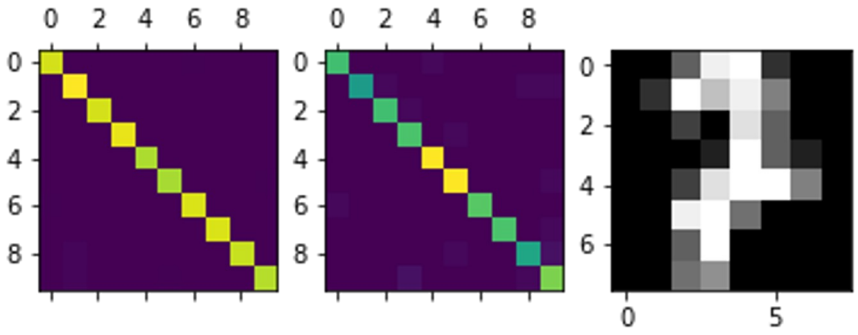
```
Training image. true label: 4, predicted label: 4.
Train accuracy: 0.9554627696590118, test accuracy: 0.9305555555555556
```



Sklearn - model_selection

```
# run a grid search to find the best parameters for the MLP classifier
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(hidden_layer_sizes=(8, 8), solver='adam', learning_rate_init=0.0001, alpha=0.0001,
max_iter=1000, activation='relu')

gs = GridSearchCV(clf, param_grid={'hidden_layer_sizes': [(8, 8), (16, 16)], 'learning_rate_init': [0.0001,
0.001], 'alpha': [0.0001, 0.001]}, cv=5, n_jobs=-1, verbose=2)

gs.fit(X_train, y_train)
best_params = gs.best_params_
best_clf = gs.best_estimator_
```

```
Training image. true label: 7, predicted label: 7.
Train accuracy: 0.9937369519832986, test accuracy: 0.9583333333333334.
Best parameters: {'alpha': 0.001, 'hidden_layer_sizes': (16, 16), 'learning_rate_init': 0.0001}
```



Pytorch

Pytorch - Tensors, randomness

- Just like numpy arrays
- Can be stored/operated on on GPU

```
x = torch.tensor([1,2,3,4,5])  #instead of numpy.array([1,2,3,4,5])
```

- Checking if cuda is available:
  - torch.cuda.is_available()
- Moving a tensor to GPU:
  - x = x.cuda()
- Moving back to CPU:
  - x = x.cpu()

- Randomness:
  - Can be controlled by setting seed:
    - torch.manual_seed(0)
  - Can create random numbers/vectors just like numpy:
    - torch.randn(2, 2)

Pytorch - Operations

- Perform operations similar to numpy arrays, Eg:
  - Element-wise addition a +b
  - Element-wise multiplication a * b
  - Matrix multiplication torch.mm(a, b.t())

- Operations can be performed on tensors on GPU

- NOTE: operations can be done only if the tensors are on the same device

Pytorch - From numpy arrays

```
x = np.array([1,2,3,4,5])
y = torch.from_numpy(x)
z = y.numpy()
```

NOTE: both the numpy array and torch tensor share the same memory location!

```
# change the array x
x[0] = 100
# This also changes y
print(y)
# tensor([100,2,3,4,5])
```

Pytorch - copy vs Inplace, Reshape, transpose, resize, etc

- y = x.add(1)
- x.add_(1)

- Just Like Numpy:
  - x.view()
  - x.reshape()
  - x.transpose()
  - x.squeeze()
  - x.unsqueeze()
  - x.resize()