



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG



CHAI

Humanities-Centered AI

Understanding Data vs. Machine Training

**Malte Luttermann – Institute for Humanities-Centered
Artificial Intelligence (CHAI)**

December 5, 2025

Overview of Contents

- (1) Programming language Python
 - (a) Introduction and first steps
 - (b) Basics
 - (c) Advanced
- (2) Markup languages
 - (a) \LaTeX
 - (b) Markdown
- (3) Development environments
 - (a) Jupyter notebooks
- (4) Version control
 - (a) Git and GitHub
- (5) Scientific computing
 - (a) NumPy and SciPy
- (6) Data processing and visualisation
 - (a) Pandas, matplotlib, and NLTK
- (7) Machine learning (scikit-learn)
 - (a) Basics (datasets, analysis)
 - (b) Simple methods (clustering, ...)
- (8) Deep learning
 - (a) PyTorch

Topics for Today

(1) Scientific computing

- NumPy
- SciPy



Acknowledgement

- The upcoming slides are taken from the following lecture and have been translated and partially modified:
 - Dr. Magnus Bender: »[Python für Machine Learning und Data Science](#)« as part of the German course »Werkzeuge für das wissenschaftliche Arbeiten«

NumPy Installation



- NumPy is a Python package
- <https://numpy.org>
- Installation, e.g., via `pip3 install numpy`
- Import via
`import numpy` or
`import numpy as np`

Why NumPy?

- Matrix and array operations (n -dimensional)
- Mathematical and numerical (standard) functions
- High performance
- Connection and usage in other packages (e.g., SciPy)



Why NumPy?

- Matrix and array operations (n -dimensional)
- Mathematical and numerical (standard) functions
- High performance
- Connection and usage in other packages (e.g., SciPy)



Internally implemented in C

NumPy Arrays

```
1 import numpy as np
2
3 one = np.array([1, 2, 3, 4, 5, 6])
4 two = np.array([
5     [1, 2, 3, 4],
6     [5, 6, 7, 8],
7     [9, 10, 11, 12]])
8
9 print(one)
10 print(two)
11
12 print(one.shape)
13 print(two.shape)
14
15 print(one[0])
16 print(two[0])
17
18 print(one[2:])
19 print(two[:,0])
```

```
$> python3 script.py
```

```
[1 2 3 4 5 6]
[[ 1 2 3 4]
 [ 5 6 7 8]
 [ 9 10 11 12]]
(6,)
(3, 4)
1
[1 2 3 4]
[3 4 5 6]
[1 5 9]
```

NumPy Arrays

```
1 import numpy as np
2
3 one = np.array([1, 2, 3, 4, 5, 6])
4 two = np.array([
5     [1, 2, 3, 4],
6     [5, 6, 7, 8],
7     [9, 10, 11, 12]])
8
9 print(one)
10 print(two)
11
12 print(one.shape)
13 print(two.shape)
14
15 print(one[0])
16 print(two[0])
17
18 print(one[2:])
19 print(two[:,0])
```

Convert list (or list of lists) to array

Array has an attribute `shape`

Access via index

Slicing as with lists, dimensions are separated by comma

ND-Arrays

```
1 import numpy as np
2
3 one = np.zeros(5)
4 two = np.ones((5,5))
5
6 print(one, one.shape)
7 print(two, two.shape)
8 print(two.ndim, two.size)
9
10 three = np.ndarray((2,2), dtype=np.float32)
11 four = np.ndarray((2,2), dtype=np.int64)
12
13 print(three, three.shape)
14 print(four, four.shape)
```

```
$> python3 script.py
[0.  0.  0.  0.  0.] (5,)
[[1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]] (5, 5)
2 25
[[1.67e-43 1.36e-43]
 [1.60e-43 1.54e-43]] (2, 2)
[[0 0]
 [0 0]] (2, 2)
```

ND-Arrays

```
1 import numpy as np
2
3 one = np.zeros(5)
4 two = np.ones((5,5))
5
6 print(one, one.shape)
7 print(two, two.shape)
8 print(two.ndim, two.size)
9
10 three = np.ndarray((2,2), dtype=np.float32)
11 four = np.ndarray((2,2), dtype=np.int64)
12
13 print(three, three.shape)
14 print(four, four.shape)
```

Array with given dimensions initialised with zeros or ones

Apart from `shape`, `ndim` and `size` attributes also exist

Empty array (random values) with given dimensions and type

Transforming Arrays

```
1 import numpy as np
2
3 one = np.arange(6)
4 print(one, one.shape)
5 two = one.reshape(3,2)
6 print(two, two.shape)
7
8 three = np.expand_dims(one, axis=0)
9 print(three, three.shape)
10
11 four = np.expand_dims(one, axis=1)
12 print(four, four.shape)
13
14 print(four[3, 0] == three[0, 3])
```

```
$> python3 script.py
```

```
[0 1 2 3 4 5] (6,)
[[0 1]
 [2 3]
 [4 5]] (3, 2)
[[0 1 2 3 4 5]] (1, 6)
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]] (6, 1)
True
```

Transforming Arrays

```
1 import numpy as np
2
3 one = np.arange(6)
4 print(one, one.shape)
5 two = one.reshape(3,2)
6 print(two, two.shape)
7
8 three = np.expand_dims(one, axis=0)
9 print(three, three.shape)
10
11 four = np.expand_dims(one, axis=1)
12 print(four, four.shape)
13
14 print(four[3, 0] == three[0, 3])
```

Bring values of array into a new form

Adding an (empty) extra dimension

Access and Filter

```
1 import numpy as np
2
3 one = np.arange(20)
4 two = np.flip(one)
5 print(one)
6 print(two)
7
8 print(one[(one % 2 == 1)])
9
10 print(one > 10)
11 print(two > 10)
12
13 print(np.nonzero(two > 10))
14 print(np.flatnonzero(two > 10))
15
16 print(one[(one > 10)])
17 print(one[(two > 10)])
```

```
$> python3 script.py
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]
[ 1  3  5  7  9 11 13 15 17 19]
[False False False False False False False False
  False False False True
  True True True True True True True True]
[ True True True True True True True True True True False
  False False
  False False False False False False False]
(array([0, 1, 2, 3, 4, 5, 6, 7, 8]),)
[0 1 2 3 4 5 6 7 8]
[11 12 13 14 15 16 17 18 19]
[0 1 2 3 4 5 6 7 8]
```

Access and Filter

```
1 import numpy as np
2
3 one = np.arange(20)
4 two = np.flip(one)
5 print(one)
6 print(two)
7
8 print(one[(one % 2 == 1)])
9
10 print(one > 10)
11 print(two > 10)
12
13 print(np.nonzero(two > 10))
14 print(np.flatnonzero(two > 10))
15
16 print(one[(one > 10)])
17 print(one[(two > 10)])
```

Filtering specific values: Boolean array selects the indices to be returned

Comparison returns a new Boolean array

Not the values but the indices

Operations

```
1 import numpy as np
2
3 a = np.array([1, 2])
4 b = np.array([1, 1])
5
6 print(a + b)
7 print(a * b)
8
9 print(a + 2)
10 print(a * 2)
```

```
$> python3 script.py
```

```
[2 3]
```

```
[1 2]
```

```
[3 4]
```

```
[2 4]
```

Operations

```
1 import numpy as np
2
3 a = np.array([1, 2])
4 b = np.array([1, 1])
5
6 print(a + b)
7 print(a * b)
8
9 print(a + 2)
10 print(a * 2)
```

Element-wise operations

Broadcasting (scalar applied to array)

Operations

```
1 A = np.array([[1, 2], [3, 4]])
2 B = np.array([[5, 6], [7, 8]])
3
4 print(A + B)
5 print(A * B)
6 print(A @ B)
7
8 print(A.sum(), A.sum(axis=1), A.sum(axis=0))
9 print(B.min(), B.max())
10
11 print(A + a) # a = np.array([1, 2])
```

```
$> python3 script.py
```

```
[[ 6  8]
 [10 12]]
[[ 5 12]
 [21 32]]
[[19 22]
 [43 50]]
10 [3 7] [4 6]
5 8
[[2 4]
 [4 6]]
```

Operations

```
1 A = np.array([[1, 2], [3, 4]])  
2 B = np.array([[5, 6], [7, 8]])  
3  
4 print(A + B)  
5 print(A * B)  
6 print(A @ B)  
7  
8 print(A.sum(), A.sum(axis=1), A.sum(axis=0))  
9 print(B.min(), B.max())  
10  
11 print(A + a) # a = np.array([1, 2])
```

Matrix multiplication

Selection of axis used for calculation

Broadcasting (array applied to array)

Example: Formula

$$d_e(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Example: Formula

$$d_e(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

```
1 import math
2
3 def d_e_py(x, y):
4     r = 0
5     for x_i, y_i in zip(x, y):
6         r += (x_i - y_i)**2
7     return math.sqrt(r)
8
9 print(d_e_py(
10     [1, 2, 1],
11     [1, 3, 2]
12 ))
```

Example: Formula

$$d_e(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

```
1 import math
2
3 def d_e_py(x, y):
4     r = 0
5     for x_i, y_i in zip(x, y):
6         r += (x_i - y_i)**2
7     return math.sqrt(r)
8
9 print(d_e_py(
10     [1, 2, 1],
11     [1, 3, 2]
12 ))
```

```
1 import numpy as np
2
3 def d_e_numpy(x, y):
4     return np.sqrt(
5         np.power(
6             (x - y),
7             2
8         ).sum()
9     )
10
11 print(d_e_numpy(
12     np.array([1, 2, 1]),
13     np.array([1, 3, 2])
14 ))
```

Example: Formula

$$d_e(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

```
$> python3 script.py
```

```
1.4142135623730951
```

```
1 import math
2
3 def d_e_py(x, y):
4     r = 0
5     for x_i, y_i in zip(x, y):
6         r += (x_i - y_i)**2
7     return math.sqrt(r)
8
9 print(d_e_py(
10     [1, 2, 1],
11     [1, 3, 2]
12 ))
```

```
1 import numpy as np
2
3 def d_e_numpy(x, y):
4     return np.sqrt(
5         np.power(
6             (x - y),
7             2
8         ).sum()
9     )
10
11 print(d_e_numpy(
12     np.array([1, 2, 1]),
13     np.array([1, 3, 2])
14 ))
```

Example: Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
```

Example: Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
```

```
$> python3 script.py
```

```
1.5134005690000123
5.295949143000001
```

Example: Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
```

```
$> python3 script.py
```

```
1.5134005690000123
5.295949143000001
```

- Time of NumPy version vs. pure Python version
- Average over 50 runs

Saving and Loading

```
1 import numpy as np
2
3 x = np.array([[1,2],[3,4]])
4 y = x.T # x.transpose()
5
6 np.save('array_x', x)
7 x_1 = np.load('array_x.npy')
8 print(x_1)
9
10 np.savetxt('array_x.txt', x)
11 x_2 = np.loadtxt('array_x.txt')
```

Saving and Loading

```
1 import numpy as np
2
3 x = np.array([[1,2],[3,4]])
4 y = x.T # x.transpose()
5
6 np.save('array_x', x)
7 x_1 = np.load('array_x.npy')
8 print(x_1)
9
10 np.savetxt('array_x.txt', x)
11 x_2 = np.loadtxt('array_x.txt')
```

```
1 np.savez('arrays', x=x, y=y)
2 with np.load('arrays.npz') as arrays:
3     print(type(arrays))
4     x_3 = arrays["x"]
5     y_3 = arrays["y"]
6
7 print(x_3, y_3)
```

Saving and Loading

```
1 import numpy as np
2
3 x = np.array([[1,2],[3,4]])
4 y = x.T # x.transpose()
5
6 np.save('array_x', x)
7 x_1 = np.load('array_x.npy')
8 print(x_1)
9
10 np.savetxt('array_x.txt', x)
11 x_2 = np.loadtxt('array_x.txt')
```

```
1 np.savez('arrays', x=x, y=y)
2 with np.load('arrays.npz') as arrays:
3     print(type(arrays))
4     x_3 = arrays["x"]
5     y_3 = arrays["y"]
6
7 print(x_3, y_3)
```

```
1 1.0000000000000000e+00 2.0000000000000000e+00
2 3.0000000000000000e+00 4.0000000000000000e+00
```

array_x.txt

Saving and Loading

```
1 import numpy as np
2
3 x = np.array([[1,2],[3,4]])
4 y = x.T # x.transpose()
5
6 np.save('array_x', x)
7 x_1 = np.load('array_x.npy')
8 print(x_1)
9
10 np.savetxt('array_x.txt', x)
11 x_2 = np.loadtxt('array_x.txt')
```

```
1 np.savez('arrays', x=x, y=y)
2 with np.load('arrays.npz') as arrays:
3     print(type(arrays))
4     x_3 = arrays["x"]
5     y_3 = arrays["y"]
6
7 print(x_3, y_3)
```

```
1 1.0000000000000000e+00 2.0000000000000000e+00
2 3.0000000000000000e+00 4.0000000000000000e+00
```

array_x.txt

```
$> python3 script.py
```

```
[[1 2]
```

```
[3 4]]
```

```
<class 'numpy.lib.npyio.NpzFile'>
```

```
[[1 2]
```

```
[3 4]]
```

```
[[1 3]
```

```
[2 4]]
```

Saving and Loading

- `np.save('file', arr)` saves a single array to a file (.npy)
- `np.savez('file', x=x, y=y)` saves multiple arrays to a file (.npz)
- `np.savetxt('file', arr)` saves an array to a text file (.txt)
 - Human-readable, but slower and larger file size
- `np.loadtxt('file')` loads an array from a text file
- `np.load('file')` loads an array (or multiple arrays) from a .npy or .npz file

Scientific Computing with SciPy – Installation



- SciPy is a Python package
- <https://scipy.org>
- Installation, e.g., via `pip3 install scipy`
- Import via
`import scipy`

Why SciPy?

- Collection of important mathematical functions
- Extension of NumPy (uses NumPy internally)
 - NumPy provides the basis (matrices and data structures)
 - SciPy provides algorithms for applications



SciPy Modules

- Fourier transforms: `scipy.fft`
- Sparse matrices: `scipy.sparse`
- Distance functions: `scipy.spatial.distance`
- Linear algebra: `scipy.linalg`
 - Similar to `numpy.linalg`
- Statistical functions: `scipy.stats`



Sparse Matrices

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sparse Matrices

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
1 import numpy as np
2
3 i_20_n = np.identity(20, dtype=
4     np.int64)
5
6 print("Number of elements",
7     i_20_n.size)
8 print("Size of element", i_20_n
9     .itemsize)
10 print("Bytes used", i_20_n.
11     nbytes)
```

Sparse Matrices

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
1 import numpy as np
2
3 i_20_n = np.identity(20, dtype=
4     np.int64)
5
6 print("Number of elements",
7     i_20_n.size)
8
9 print("Size of element", i_20_n
10     .itemsize)
11
12 print("Bytes used", i_20_n.
13     nbytes)
```

```
1 from scipy.sparse import(
2     bsr_array, coo_array,
3     csc_array, csr_array,
4     dia_array, dok_array,
5     lil_array
6 )
7
8 from scipy.sparse import identity
9
10 i_20_s = identity(20, dtype=np.int64,
11     format='dia')
12
13 print("Number of elements", i_20_s.
14     data.size + i_20_s.offsets.size)
15
16 print("Size of element", i_20_s.data.
17     itemsize, i_20_s.offsets.itemsize)
18
19 print("Bytes used", i_20_s.data.nbytes
20     + i_20_s.offsets.nbytes)
```

Sparse Matrices

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
1 import numpy as np
2
3 i_20_n = np.identity(20, dtype=
4     np.int64)
5
6 print("Number of elements",
7     i_20_n.size)
8 print("Size of element", i_20_n
9     .itemsize)
10 print("Bytes used", i_20_n.
11     nbytes)
```

```
1 from scipy.sparse import (
2     bsr_array, coo_array,
3     csc_array, csr_array,
4     dia_array, dok_array,
5     lil_array
6 )
7
8 from scipy.sparse import identity
9
10 i_20_s = identity(20, dtype=np.int64,
11     format='dia')
12
13 print("Number of elements", i_20_s.
14     data.size + i_20_s.offsets.size)
15 print("Size of element", i_20_s.data.
16     itemsize, i_20_s.offsets.itemsize)
17 print("Bytes used", i_20_s.data.nbytes
18     + i_20_s.offsets.nbytes)
```

```
$> python3 script.py
```

```
Number of elements 400
Size of element 8
Bytes used 3200
Number of elements 21
Size of element 8 4
Bytes used 164
```

Sparse Matrices

- Different formats for sparse matrices depending on the distribution of zeros and the required functions

```
1 from scipy.sparse import (  
2     bsr_array, coo_array,  
3     csc_array, csr_array,  
4     dia_array, dok_array,  
5     lil_array  
6 )
```

- `np.identity(20, dtype=np.int64)` stores each zero as a number
- `scipy.sparse.identity(20, dtype=np.int64, format='dia')` stores only the values different from zero and their positions
- Documentation: <https://docs.scipy.org/doc/scipy/reference/sparse.html>

Statistical Functions

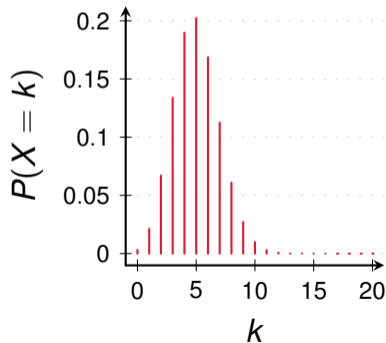
- Example: Binomial distribution

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

Statistical Functions

- Example: Binomial distribution

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

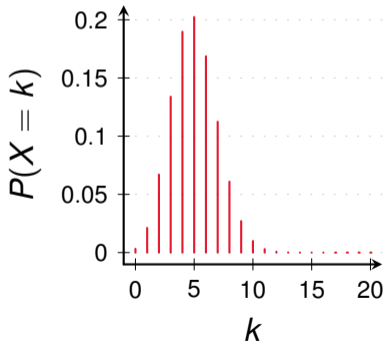


Statistical Functions

■ Example: Binomial distribution

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

```
1 from scipy.stats import binom
2
3 n, p = 20, 0.25
4 mean, variance = binom.stats(n, p, moments='mv')
5
6 print(mean, variance)
```



Statistical Functions

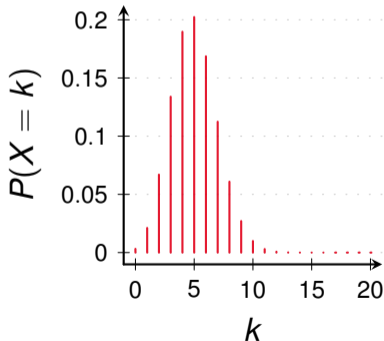
- Example: Binomial distribution

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

```
1 from scipy.stats import binom
2
3 n, p = 20, 0.25
4 mean, variance = binom.stats(n, p, moments='mv')
5
6 print(mean, variance)
```

```
$> python3 script.py
```

```
5.0 3.75
```



Again: Example Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
12
13 from scipy.spatial import distance
14
15 print(timeit.timeit('distance.euclidean(x, y)', number=50, globals=
    globals()))
```

Again: Example Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
12
13 from scipy.spatial import distance
14
15 print(timeit.timeit('distance.euclidean(x, y)', number=50, globals=
    globals()))
```

```
$> python3 script.py
```

```
1.5134005690000123
5.295949143000001
0.12705923099998984
```

Again: Example Runtime

An implementation using NumPy is already faster, but SciPy is even more optimised!

```
$> python3 script.py
```

```
1.5134005690000123  
5.295949143000001  
0.12705923099998984
```

```
1 import timeit  
2  
3 x = np.random.rand(1000000)  
4 y = np.random.rand(1000000)  
5  
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))  
7  
8 x_list = x.tolist()  
9 y_list = x.tolist()  
10  
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=  
    globals()))  
12  
13 from scipy.spatial import distance  
14  
15 print(timeit.timeit('distance.euclidean(x, y)', number=50, globals=  
    globals()))
```

Again: Example Runtime

```
1 import timeit
2
3 x = np.random.rand(1000000)
4 y = np.random.rand(1000000)
5
6 print(timeit.timeit('d_e_numpy(x, y)', number=50, globals=globals()))
7
8 x_list = x.tolist()
9 y_list = x.tolist()
10
11 print(timeit.timeit('d_e_py(x_list, y_list)', number=50, globals=
    globals()))
12
13 from scipy.spatial import distance
14
15 print(timeit.timeit('distance.euclidean(x, y)', number=50, globals=
    globals()))
```

An implementation using NumPy is already faster, but SciPy is even more optimised!

```
$> python3 script.py
```

```
1.5134005690000123
5.295949143000001
0.12705923099998984
```

Uses `numpy.linalg.norm` internally

Summary

- Scientific computing
 - NumPy
 - SciPy

