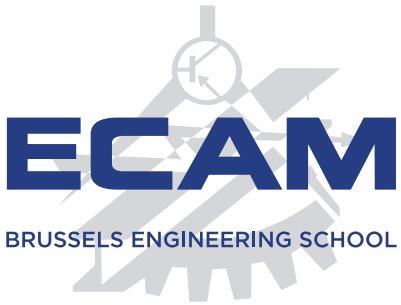


NC301μ Introduction to
Numerical Computing with SciPy

Session 1

Multidimensional Arrays and Linear Algebra with NumPy

Sébastien Combéfis

Winter 2020

Objectives

- Understand what is numerical computing

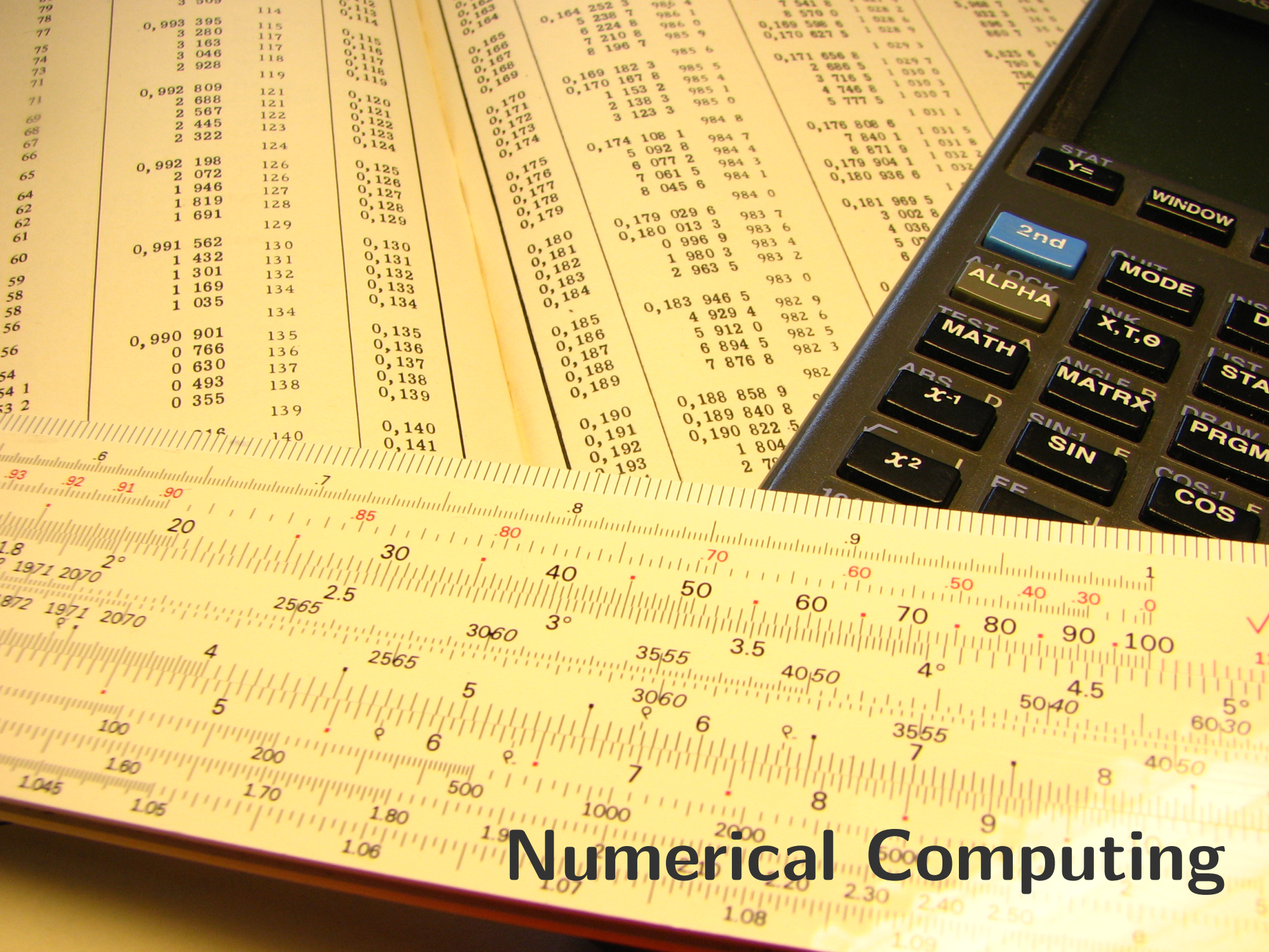
  And when and how it can be used to solve real life problems

- Use the multidimensional arrays of NumPy to store data

  And discover the operations that can be applied on them

- Use matrices and operations to solve linear algebra problems

  Thanks to the `numpy` module of the SciPy ecosystem


Numerical Computing

Numerical Computing

- Numerical computing studies algorithms to solve problems
  - Combination of computer science and mathematics fields
  - For problems in science, engineering, medicine, business, etc.
- Numerical analysis uses numerical approximations

  Instead of having a solution based on symbolic manipulations

Numerical Computing Characteristics

- Numerical computing uses the power of computer

  Microelectronic revolution lead to powerful machines

- Three main characteristics specific to numerical computing
  - **Accuracy**: how large is the computation error?

    Inexact algorithms, limited floating-point number representation
  - **Efficiency**: how fast are the results produced?

    Time and space complexities and human time to model problem
  - **Numerical stability**: how are errors propagating?

    Possible exponential growth of errors when propagating

Numerical Computing Process

- Used for problems that can be solved with mathematics

  First step is to formulate the problem with a mathematical model

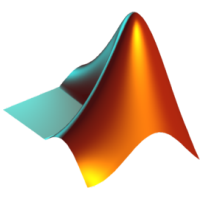
- Computer is then used to find a solution to the problem
  - The appropriate numerical method must be selected
  - And then the method should be implemented as a program
- Finally, the obtained solution must be validated

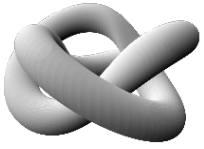
  The computer will just output a value that should be checked

Numerical Computing Tools

- Several kinds of tools for numerical computing do exist
  - "Toolboxes" application used as a blackbox for the scientist
  - Programming languages to write "any" application
- Programming languages are the most flexible solutions

  Specific and dedicated languages or librairies for languages

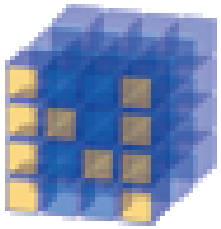




SciPy Ecosystem

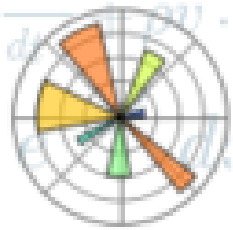
- Softwares ecosystem for mathematics, science and engineering

  Collection of several coordinated libraries



NumPy

SciPy



Matplotlib

IPython

SymPy

pandas

Multidimensional Array

numpy Library

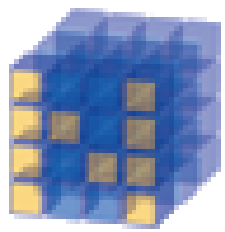
- Base library for numerical computing

  Mostly written in C and Python, integrates Fortran

- Many features supported by this package
  - Representing multidimensional arrays (N dimensions)
  - Sophisticated functions
  - Linear algebra, Fourier transform
  - ...

- Open source code available on GitHub

  https://github.com/numpy/numpy



ndarray Object

- Multidimensional arrays represented with the `ndarray` object

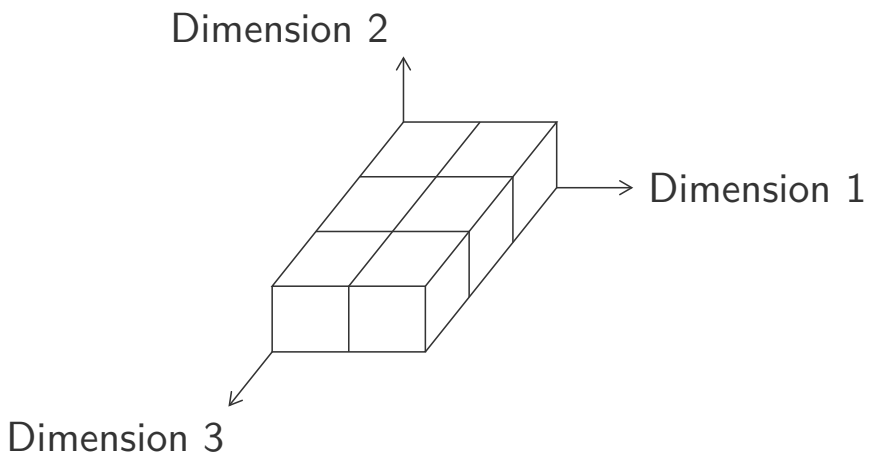
  Data structure at the heart of numerical computing with SciPy

- Characterised by a shape represented as a tuple

  Specifies the number of elements on each dimension

```
1  shape = (2, 1, 3)
2  data = np.ndarray(shape)
3
4  print(type(data))
5  print(data.ndim)
6  print(data.size)
```

```
<class 'numpy.ndarray'>
3
6
```



Creating ndarray

- Randomly initialised `ndarray` obtained with `empty` function

```
1 a = np.empty((1, 3))
```

- Multidimensional arrays can be created from Python list

```
1 a = np.array([1, 2, 3])
2 b = np.array([[1, 2, 3], [4, 5, 6]])
```

- Creating value initialised `ndarray` with specific functions

```
1 a = np.zeros((2))
2 b = np.ones((1, 2))
3 c = np.full((1, 3), 0.5)
```

Incremental Sequence

- Specific functions to create incremental sequences

  Easier way to create one-dimensional `ndarray` with specific values

- First and last elements and possible step with `arange` function

```
1 a = np.arange(7)
2 b = np.arange(2, 7)
3 c = np.arange(2, 7, 2)
```

- First and last elements and size with `linspace` function

```
1 a = np.linspace(2, 7, 6)
2 b = np.linspace(2, 7, 6, endpoint=False)
```

Accessing Element

- Elements from an `ndarray` are accessed with access operator

  Specifying the position of the element in each dimension

- Subarray obtained by not giving indexes for each dimension

  Extracted array has less dimensions than the original

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

```
4
[[1]
 [2]]
[4]
```

Slice

- Subarray can also be obtained with the slicing operator

  Specifying ranges of elements to extract in each dimension

```
1  data = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])
2
3  print(data[1:3,1:4])
4  print(data[0,:])
5  print(data[:,0])
6  print(data[0:1,[0, 2, 3]])
7  print(data[[0, 1, 2],[0, 3, 1]])
```

```
[[ 5  6  7]
 [ 9 10 11]]
[0 1 2 3]
[0 4 8]
[[0 2 3]]
[0 7 9]
```

ndarray Modification

- Elements of an `ndarray` modified with the access operator

  Either a single element or multiple elements at the same time

```
1  data = np.array([[0, 1, 2], [3, 4, 5]])
2  data[1,0] = -1
3  print(data)
4
5  data[:,2:] = np.full((2, 1), -2)
6  print(data)
```

```
[[ 0  1  2]
 [-1  4  5]]
[[ 0  1 -2]
 [-1  4 -2]]
```

Mask

- An `ndarray` with boolean values is called a mask

  Created with boolean values or from boolean expression

- A mask is used to extract some elements from an `ndarray`

  Elements for which mask value is `True` at the same position

```
1 data = np.array([[-1, 4, 0], [-3, 2, 1]])
2 print(data[data % 2 == 0])
```

```
[4 0 2]
```

Reshaping

- Possible to change the shape of an `ndarray`

  While keeping exactly the same elements but organised differently

```
1  a = np.arange(6)
2  print(a.shape)
3  print(a)
4
5  b = a.reshape(2, 3)
6  print(b.shape)
7  print(b)
```

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

Operation

- Possible to manipulate `ndarray` objects with operations

  Two main ways to execute the operations on the elements

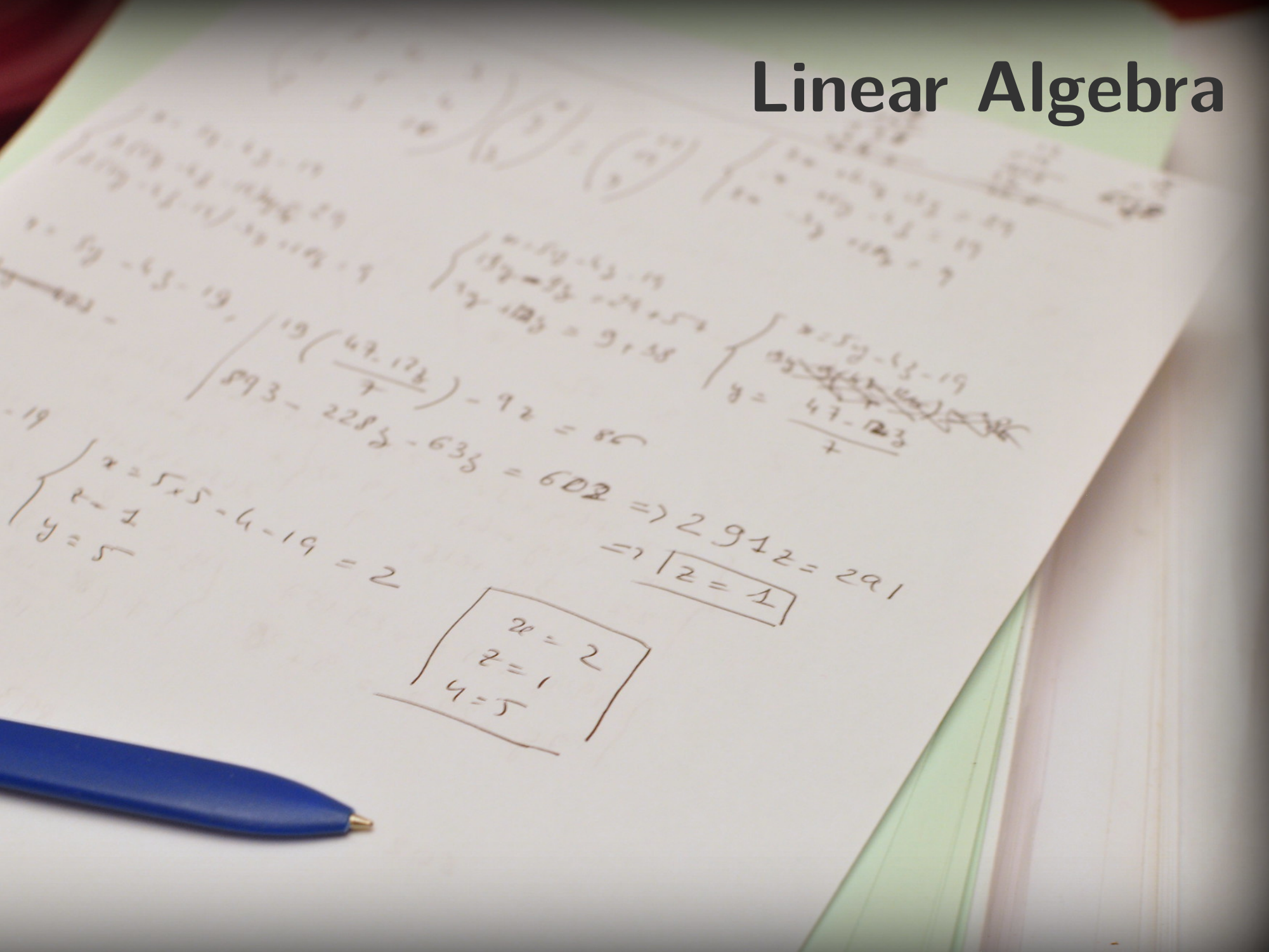
- Element-wise operations for compatible `ndarray`

```
1  a = np.array([[1, -1], [2, 3]])
2  b = np.arange(4).reshape(2, 2)
3  c = a + b
```

- Vectorial expression to apply same operation to all elements

```
1  a = np.arange(6).reshape(2, 3)
2  b = data ** 2
3
4  x = np.arange(10)
5  data = np.sin(x) ** 2 + np.cos(x) ** 2
```

Linear Algebra



matrix Object

- Matrix is a two dimensional array represented by `matrix`

  Specific object with specific methods and operations available

```
1  mat = np.matrix([[1, 2, 3], [4, 5, 6]])
2  print(mat)
3  print(mat.shape)
4
5  print(np.matrix('1 2;3 4;5 6'))
```

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

Creating matrix

- Identity matrix created with `identity` or `eye` function

```
1 a = np.identity(3)
2 b = np.eye(2)
3 c = np.eye(2, 4, 1)
```

- Possible to create triangular matrix with `tri` function

```
1 a = np.tri(2)
2 b = np.tri(2, 4, 1)
```

- Converting a one dimensional array to diagonal matrix

```
1 a = np.array([4, 0, -2])
2 b = np.diag(a)
```

Vector Operation

- Several operations specific to vectors are supported

  Usual arithmetic, norm, scalar and vectorial products

```
1  u = np.array([1, 2, 3])
2  v = np.array([4, 5, 6])
3
4  print(2 * u + v)
5  print(norm(u))
6  print(np.vdot(u, v))
7  print(np.cross(u, v))
8  print(u * v)
```

```
[ 6  9 12]
3.7416573867739413
32
[-3  6 -3]
[ 4 10 18]
```

Matrix Properties

- Properties from matrix obtained with specified functions

  Dimension, main diagonal, trace, rank, determinant

```
1  A = np.matrix('1 2 3;4 5 6')
2  B = np.identity(2)
3  C = np.matrix('1 2 3;2 4 6;3 6 9')
4
5  print(A.shape)
6  print(np.diag(A))
7  print(np.trace(B))
8  print(matrix_rank(A))
9  print(det(C))
```

```
(2, 3)
[1 5]
2.0
2
0.0
```

Matrix Operation (1)

- Several operations specific to matrices are supported

  Usual arithmetic, transpose, product, inverse

```
1  A = np.matrix('1 2 3;4 5 6')
2  B = np.identity(2)
3
4  print(2 * A[:,:2] + B)
5  print(A.T)
```

```
[[ 3.  4.]
 [ 8. 11.]]
[[1 4]
 [2 5]
 [3 6]]
```

Matrix Operation (2)

- Several operations specific to matrices are supported

  Usual arithmetic, transpose, product, inverse

```
1  C = np.diag(np.array([1, 2, 3]))
2  D = np.matrix('1 2 3;2 4 6;3 6 9')
3  E = np.mat(np.array([[1, 2], [3, 4]]))
4
5  print(D * C[:,:2])
6  print(E.I)
```

```
[[ 1  4]
 [ 2  8]
 [ 3 12]]
[[-2.   1. ]
 [ 1.5 -0.5]]
```

References

- Mohammed Amarnah (2018). *Computational Numerical Analysis*, June 23, 2018. https://medium.com/@m.amarnah/computational-numerical-analysis-517eda19cef8
- Martin Heller (2018). *What is Julia? A fresh approach to numerical computing*, InfoWorld, June 27, 2018. https://www.infoworld.com/article/3284380/what-is-julia-a-fresh-approach-to-numerical-computing.html
- Vasudev (2017). *Introduction to Numpy -1 : An absolute beginners guide to Machine Learning and Data science*, September 28, 2017. https://hackernoon.com/introduction-to-numpy-1-an-absolute-beginners-guide-to-machine-learning-and-data-science-5d87f13f0d51
- Akshay Pal (2019). *Linear Algebra and Numpy*, August, 10, 2019. https://medium.com/secure-and-private-ai-math-blogging-competition/linear-algebra-and-numpy-ab18214fd752

Credits

- Logos pictures from Wikipedia.
- fdecomite, May 8, 2009, https://www.flickr.com/photos/fdecomite/3511830891.
- Bernd Hutschenreuther, December 25, 2018, https://www.flickr.com/photos/116228447@N06/32789894838.
- Frédérique Voisin-Demery, December 22, 2014, https://www.flickr.com/photos/vialbost/16084697841.