Informatics 3

Ádám Gyenge

Course information

Format

- Lecture: Tuesday 8.30-10.00
- Webpage:
  `https://adamgyenge.gitlab.io/teaching/info3/2026/`
- Lecture notes on the website (work in progress, be aware of mistakes)
- Email: `Gyenge.Adam@ttk.bme.hu`

Content

1. Scientific programming in Python
   - Advanced features of NumPy
   - Symbolic computations with SymPy
   - An outlook to SAGE
   - Methods of collaboration: GIT, Scrum
2. Computational topology
   - Basics of topology
   - Knots and links
   - 2-manifolds
   - Triangulations and simplicial complexes

Final grade

1. Midterm 1 (on week 6 lecture): 30%
2. Midterm 2 (on week 12 lecture): 30%
3. Project: 30%
   - Task: solve an actual scientific problem using SymPy (and possibly other Python libraries)
   - Some ideas are given in Section 7 of the notes
   - Output: Jupyter notebook or latex document (+Python source code), about 3-4 pages [A4]
   - Can be done in pairs (then 6-8 pages)
   - Presentation of ideas (2-3 mins): week 6 lab
   - Final presentation (10-15 mins): week 13 lab
4. Participation: 10%

Introduction to Python in Science

- Python is an open-source, high-level programming language.
- Widely adopted in scientific computing for its simplicity and versatility.
- Offers extensive libraries for data analysis, visualization, and computation.

Why Python?

- Easy to learn and use.
- Strong community support.
- Cross-platform compatibility.

Core Libraries in the Ecosystem

Popular Libraries

- **NumPy**: Numerical computations with multi-dimensional arrays.
- **SciPy**: Advanced scientific computing.
- **Pandas**: Data manipulation and analysis.
- **Matplotlib** and **Seaborn**: Data visualization.
- **SymPy**: Symbolic mathematics.

Introduction to NumPy

- NumPy (Numerical Python) is an open-source library for numerical computing in Python.
- Created in 2005 by Travis Oliphant by merging features from two predecessor libraries: Numeric and Numarray.
- Introduced a unified and efficient array object for advanced mathematical operations.
- Serves as the basis for many other libraries, including SciPy, pandas, and scikit-learn.
- Widely used in fields such as data analysis, machine learning, and scientific research.

Core Technology

- NumPy leverages optimized libraries like BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage).
- BLAS provides low-level routines for vector and matrix operations.
- LAPACK builds on BLAS for complex problems, including solving linear systems and eigenvalue computations.
- Both BLAS and LAPACK are written in highly optimized C and Fortran, ensuring speed and reliability.
- This reliance on optimized libraries makes NumPy a cornerstone of high-performance scientific computing.

Key Features of NumPy

- Efficient multi-dimensional array object (`ndarray`).
- Broad range of mathematical functions.
- Broadcasting and vectorization for performance.

Installing and importing NumPy

1. Install SymPy using pip to get started:

   `pip install numpy`

2. Import SymPy into your Python script as `np`:

   `import numpy as np`

Creating Arrays

The key data type in NumPy is that of an N-dimensional array object, called `ndarray`.

```
# Vector and matrix
v = np.array([1, 2, 3])
A = np.array([[1, 2], [3, 4]])
# Random matrix
B = np.random.random((3, 3))
```

- Vectors: 1D arrays.
- Matrices: 2D arrays.
- Arrays can be initialized from lists or randomly.

Properties of arrays

- **Shape**: Specifies the dimensions of the array (e.g., rows and columns). Accessed using `array.shape`.
- **Data Type (dtype)**: Defines the type of elements in the array, such as integers, floats, or complex numbers. Accessed using `array.dtype`.
- **Size**: Total number of elements in the array. Accessed using `array.size`.
- **Dimension (ndim)**: Indicates the number of dimensions (axes) of the array. Accessed using `array.ndim`.
- **Item Size**: Memory size (in bytes) of each array element. Accessed using `array.itemsize`.
- **Memory Layout**: Arrays can be stored in row-major (C-style) or column-major (Fortran-style) order. Accessed using `array.flags`.
- **Mutability**: NumPy arrays are mutable, meaning their contents can be modified after creation.
- **Homogeneity**: All elements in a NumPy array must be of the same data type for efficient computation.

Element-wise Operations

```
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
result = v1 + v2
```

- Supports element-wise addition, subtraction, multiplication, etc.

Matrix Multiplication and Transpose

```
C = np.dot(A, B)
A_T = np.transpose(A)
```

- Use `np.dot()` for matrix multiplication.
- Transpose matrices using `np.transpose()`.

Submatrices

```
submatrix = B[1:, 1:]
column_vector = A[:, 0]
```

- Extract specific parts of matrices.
- Useful for analyzing large datasets.

Vector and Matrix Norms

```
vector_norm = np.linalg.norm(v)
matrix_norm = np.linalg.norm(A, 'fro')
```

- Measure size or magnitude.
- Vector norms: Length of a vector.
- Frobenius norm: Matrix magnitude.

The result are of type `np.float64`:

```
>>> vector_norm
np.float64(3.7416573867739413)

>>> matrix_norm
np.float64(3.872983346207417)
```

Solving Linear Equations

```
A = np.array([[2, 1], [1, -3]])
b = np.array([8, 1])
x = np.linalg.solve(A, b)
```

- Solve $Ax = b$ using `np.linalg.solve()`.

Result:

```
>>> x
array([3.57142857, 0.85714286])
```

Eigenvalues and Eigenvectors

```
# Finding eigenvalues and eigenvectors
A = np.array([[4, -2],
              [1, 1]])
eigenvalues, eigenvectors = np.linalg.eig(A)
```

Result:

```
>>> eigenvalues
array([3., 2.])

>>> eigenvectors
array([[0.89442719, 0.70710678],
       [0.4472136 , 0.70710678]])
```

SVD Decomposition

```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
U, S, VT = np.linalg.svd(A)
```

This gives:

```
>>> U
array([[-0.2298477 ,  0.88346102,  0.40824829],
       [-0.52474482,  0.24078249, -0.81649658],
       [-0.81964194, -0.40189603,  0.40824829]])
>>> S

array([9.52551809, 0.51430058])

>>> VT
array([[-0.61962948, -0.78489445],
       [-0.78489445,  0.61962948]])
```

QR Decomposition

QR decomposition: a matrix A is decomposed into an orthogonal matrix Q and an upper triangular matrix R, such that

$$A = QR$$

In NumPy:

```
A = np.array([[1, 2, 4], [3, 8, 14], [2, 6, 13]])

# Perform QR decomposition
Q, R = np.linalg.qr(A)
```

Broadcasting

- One of the most powerful features of NumPy is broadcasting, which allows arrays of different shapes to be used in arithmetic operations.
- Instead of reshaping the arrays manually, NumPy automatically stretches the smaller array along the missing dimensions.

```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
B = np.array([1, 2, 3])
```

The we can add this 1D array to each row of the matrix

```
>>> A+B
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12]])
```

Vectorization

- Replace loops with array operations.
- Uses optimized C code in the background instead of Python loops
- Drastically speeds up computations.

Example:

```
data = np.random.random(1000000)
squared_data = data ** 2
```

Masked Arrays

- Masked arrays are arrays that allow elements to be masked or ignored during calculations.
- This is useful in scientific datasets where missing or invalid data may occur.

```
# Create an array with invalid data
data = np.array([1, 2, -999, 4, 5])

# Mask the invalid data (-999)
masked_data = np.ma.masked_values(data, -999)
```

The result looks in Python as follows.

```
>>> masked_data
masked_array(data=[1, 2, --, 4, 5],
             mask=[False, False,  True, False, False],
       fill_value=-999)
```

Masked Arrays

One we have a masked array, we can perform various calculations on it. For example, let us compute the mean of the data set, excluding the masked elements:

```
# Calculate the mean, ignoring the masked element
>>> masked_data.mean()
np.float64(3.0)
```

Masked arrays are particularly important in fields like astronomy and climate science, where datasets often have missing or invalid entries due to sensor errors or data corruption.

Memory Mapping

- NumPy supports memory mapping of large arrays stored in binary files on disk, allowing for partial loading of the data without loading the entire dataset into memory.
- This feature is useful when working with extremely large datasets that cannot fit into the available memory.
- Instead of loading the entire array, NumPy accesses only the required sections, making computations possible on memory-constrained systems.

```
filename = 'data.dat'
large_array = np.memmap(filename, dtype='float32',
    mode='w+', shape=(10000, 10000))

# Assign values to parts of the array
large_array[:1000, :1000] = np.random.random((1000, 1000))

# Flush changes to disk
large_array.flush()
```

Structured Arrays

- NumPy also supports structured arrays, which allow users to store heterogeneous data (e.g., mixed types) in a single array.
- Structured arrays can be thought of as NumPy's version of a database table or a spreadsheet, where each column can have different types.

```
# Define a structured data type with fields
dt = np.dtype([('name', 'U10'), ('age', 'i4'),
    ('weight', 'f4')])

# Create a structured array
people = np.array([('Alice', 25, 55.0),
    ('Bob', 30, 85.5)], dtype=dt)

print("Names:", people['name'])
print("Ages:", people['age'])
print("Weights:", people['weight'])
```

Advanced Indexing

- In addition to basic slicing, NumPy supports advanced indexing techniques such as boolean indexing and indexing with integer arrays.
- These techniques are useful when selecting specific subsets of data based on conditions or patterns.

```
# Create an array of numbers
data = np.array([10, 20, 30, 40, 50])

# Boolean indexing: select elements greater than 30
greater_than_30 = data[data > 30]
```

In-place Operations

```
arr = np.array([1, 2, 3])
arr += 10
```

- Modify arrays without creating new ones.

Numerical Methods with SciPy

SciPy Overview:

- SciPy is built on NumPy for high-level scientific computations.
- Provides modules for integration, differentiation, optimization, and more.
- Commonly used in scientific computing, engineering, and data analysis.

Modules Discussed:

- Integration: `scipy.integrate`
- Optimization: `scipy.optimize`
- Signal/Image Processing: `scipy.signal`, `scipy.ndimage`
- Linear Algebra: `scipy.linalg`
- Statistics: `scipy.stats`

Numerical Integration

Integration with SciPy:

- `scipy.integrate` provides functions for definite and indefinite integrals.
- Example: Definite integral of x^2 from 0 to 1.

```
def func(x):
    return x**2

result, error = sci.integrate.quad(func, 0, 1)
print(result, error)
```

Output:

```
>>> result
0.3333333333333333

>>> error
3.700743415417188e-15
```

Solving Differential Equations

- `solve_ivp` provides tool solving DE's numerically
- Example: Solve $dy/dt = -2y$.

```
# Define the differential equation dy/dt = -2y
def dydt(t, y):
    return -2 * y

# Solve the equation with initial condition y(0) = 1
solution = integrate.solve_ivp(dydt, [0, 5], [1],
   method='RK45', t_eval=np.linspace(0, 5, 100))
```

Arguments:

- Time span $[t_0, t_{end}]$ for the solution is $[0, 5]$
- Initial condition is set to be $y(0) = 1$.
- The argument `method='RK45'` specifies the Runge-Kutta method for integration.
- The argument `t_eval` gives the time points at which to store the solution.

Result

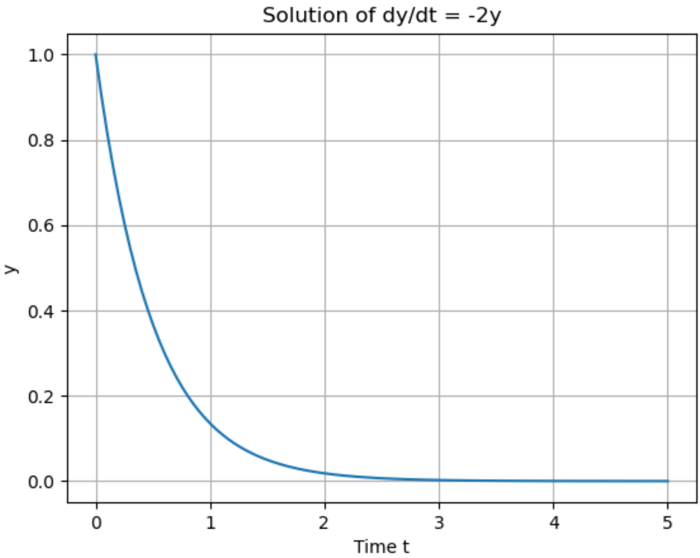

Image Processing

Image Manipulation:

- `scipy.ndimage` provides tools for image filtering and transformations.
- Example: Apply Gaussian blur.

```
blurred_image = ndimage.gaussian_filter(image, sigma=2)
```

Use Cases:

- Smoothing images.
- Edge detection.

Statistics with `scipy.stats`

- Probability theory and statistics.
- Distributions include:
  - **Continuous**: Normal (`norm`), Exponential (`expon`), Uniform (`uniform`), Beta (`beta`), etc.
  - **Discrete**: Binomial (`binom`), Poisson (`poisson`), Geometric (`geom`), etc.

Example:

```
from scipy.stats import norm

# Probability density function (PDF)
x = norm.pdf(0, loc=0, scale=1)

# Cumulative distribution function (CDF)
y = norm.cdf(0, loc=0, scale=1)

# Generate random samples
samples = norm.rvs(size=1000)
```

Basic Statistics in `scipy.stats`

- **Descriptive Statistics:** `mean`, `median`, `mode`, `variance`, `std`.
- **Order Statistics:** `percentileofscore`, `scoreatpercentile`.
- **Moments:** `moment` (e.g., skewness (3rd), kurtosis (4th)).

Example:

```
from scipy.stats import skew, kurtosis
import numpy as np

# Generate data
data = np.random.normal(size=100)

# Compute statistics
mean = np.mean(data)
skewness = skew(data)
kurt = kurtosis(data)
```

Advanced Statistics with scipy.stats

- **Hypothesis Testing:** ttest_ind, ttest_rel, chi2_contingency, ks_2samp.
- **Correlation Analysis:** pearsonr, spearmanr.
- **Fit to Data:** curve_fit, kde.
- **ANOVA:** f_oneway for one-way ANOVA.

Example:

```
from scipy.stats import ttest_ind, pearsonr

# Two-sample t-test
result = ttest_ind([1, 2, 3], [4, 5, 6])

# Correlation coefficient
corr, p_value = pearsonr([1, 2, 3], [1, 2, 4])
```

Linear Algebra

Eigenvalues and Eigenvectors:

- `scipy.linalg` extends NumPy for advanced linear algebra.
- Example: Compute eigenvalues and eigenvectors.

```
eigenvalues, eigenvectors = scipy.linalg.eig(matrix)
print(eigenvalues, eigenvectors)
```

LU Decomposition

Matrix Factorization:

- Decompose a matrix into P, L, U.
- Useful for solving linear systems.

```
P, L, U = scipy.linalg.lu(matrix)
print(L, U)
```

Sparse Matrices

A **sparse matrix** or **sparse array** is a matrix in which most of the elements are zero.

Efficient Matrix Representation:

- `scipy.sparse` for handling large, sparse datasets.
- Example: Create and manipulate sparse matrices.

```
sparse_matrix = csr_matrix(dense_matrix)
transpose = sparse_matrix.transpose()
```