18YZALG — Tutorial 7 Assignments

Tutorial 10 — Scientific Algorithmization Mini-Projects

Choose **one** mini-project per group and present your solution in the tutorial session.

Quick facts

Item	Description
Who	Student groups (recommended 2-3 students)
What	Short demo sent via email
We practice	Taylor expansion, interpolation, numerical differentiation and integration, numerical linear algebra, PCA, error checks, and small measurements
Grading	Credit to the continuous assessment (tutorial part of the course)

What every group must include

- **Problem statement**: input, output, edge cases, and why the task is interesting.
- **Numerical model**: what is the mathematical object? Function, table, matrix, sampled signal, or data matrix?
- **At least two approaches** to compare. One should be the recommended method; the other can be a baseline or simpler method.
- **Correctness checks**: a few small `assert` tests are enough. Include at least one edge case.
- **Error or quality measurement**: for example absolute error, residual, reconstruction error, or comparison with a known exact answer.
- **Small runtime or size measurement**: show what changes when the number of terms, samples, rows, columns, or components changes.
- **Conclusion**: what method would you recommend, and under which assumptions?

Submission format

- Send all relevant files to buresj11@fjfi.cvut.cz
- Use slides, a notebook, terminal output, printed examples, or a small visualization - anything is fine as long as the required content is covered.

Assessment (20 points)

Criterion	Points	What we look for
Numerical model and correctness	0-5	Inputs, outputs, assumptions, and edge cases are clear; solution actually works
Algorithm choice and explanation	0-5	The chosen method matches the problem; parameters are explained
Tests and measurement	0-5	Tests are shown; error or quality measurement is interpreted honestly
Demo clarity	0-5	Clear structure, readable output, good pacing, and a practical conclusion

Mini-projects

1. Taylor tolerance calculator

Goal. Build a small calculator that approximates a function using a Taylor expansion and automatically decides how many terms are needed to reach a requested tolerance. The simplest version should approximate `exp(x)` around `0`.

This project shows the practical meaning of a local approximation: the same method may work very well near the expansion point and require many more terms farther away.

Required content

- Implement Taylor approximation for `exp(x)` around `0`.
- Add a function that keeps adding terms until the newest term is smaller than a tolerance such as `1e-6`.
- Compare your approximation with `math.exp(x)` for several values of `x`.
- Print the number of terms used for each `x`.
- Include an edge case such as `x = 0`.
- Explain why the cost grows with the number of terms.

Suggested approaches to compare

- Method A: fixed number of terms, for example 5 or 10.
- Method B: automatic stopping based on tolerance.
- Optional Method C: implement `sin(x)` using its Taylor expansion.

Demo focus

- Show a table with columns: `x`, approximation, exact value, error, and number of terms.
- Show that points farther from `0` usually need more terms.
- Show one plot of error versus number of terms.

Stretch goals (optional)

- Let the user choose the tolerance.
- Add a maximum number of terms and explain what happens when the tolerance is not reached.
- Compare direct factorial computation with an incremental recurrence for the next term.

2. Sensor table interpolation

Goal. You have a calibration table from a sensor. For example, temperature values are known only at selected points. Given a new input value between two table entries, estimate the corresponding output.

This project shows how to turn a table of measured values into a simple predictive function.

Required content

- Store a small calibration table as two arrays: `xs` and `ys`.
- Implement nearest-neighbor interpolation.
- Implement piecewise linear interpolation.
- Test at least three query values, including a query exactly equal to a known `x` value.
- Decide what your program does when the query is outside the table range.
- Compare your result with `numpy.interp`.

Suggested approaches to compare

- Method A: nearest neighbor.
- Method B: piecewise linear interpolation.
- Optional Method C: polynomial fitting with `numpy.polyfit` and a warning about overfitting or wiggles.

Demo focus

- Plot the known sample points and the interpolated curve.
- Print a small table with query values and predictions from both methods.
- Explain why linear interpolation is often a safe first choice.

Stretch goals (optional)

- Load the calibration table from a text file.
- Add input validation for unsorted `xs`.
- Compare interpolation error on a known function such as `sin(x)`.

3. Motion from samples: speed and distance

Goal. A moving object is measured at discrete times. You know position values `s(t)` at sampled time points. Estimate speed using numerical differentiation and estimate travelled distance using numerical integration.

This project connects two tutorial topics: slopes and areas from sampled data.

Required content

- Create or load a sampled position signal.
- Estimate speed using forward difference and central difference.
- Estimate travelled distance by integrating speed with the trapezoid rule.
- Include a test signal where the exact derivative or exact integral is known.
- Compare error for several step sizes or sample counts.
- Explain why very sparse sampling gives a poor estimate.

Suggested approaches to compare

- Method A: forward difference for speed.
- Method B: central difference for speed.
- Optional Method C: compare trapezoid integration with a simple rectangle rule.

Demo focus

- Plot position versus time.
- Plot estimated speed versus exact speed for a known example.
- Show a table of error as the number of samples increases.

Stretch goals (optional)

- Add noisy measurements and smooth them with a moving average before differentiating.
- Estimate acceleration as a second derivative.
- Use real-looking generated data, for example a bicycle ride with starts and stops.

4. Noisy model fitting with least squares

Goal. You have noisy measurement pairs `(x, y)`. Fit a simple linear model `y = ax + b` and explain how good the fit is using residuals.

This project introduces numerical linear algebra in a very practical way: build a matrix, solve a least-squares problem, and interpret the result.

Required content

- Generate or load noisy data points.
- Build the design matrix for a line: columns `x` and `1`.
- Use `numpy.linalg.lstsq` to estimate slope and intercept.
- Compute predictions and residuals.
- Print the fitted equation and the mean squared error.
- Include a small exact test where all points lie on one line.

Suggested approaches to compare

- Method A: fit a constant model that predicts only the mean of `y`.
- Method B: fit a line using least squares.
- Optional Method C: fit a quadratic model and compare whether the extra complexity helps.

Demo focus

- Plot measured points and the fitted line.
- Show residuals or at least the mean squared error.

- Explain why an overdetermined system usually cannot be solved exactly.

Stretch goals (optional)

- Add outliers and discuss how they affect least squares.
- Compare training error and test error by splitting the data.
- Fit a polynomial of degree 2 or 3 and warn about overfitting.

5. PCA compression explorer

Goal. Create a small two-dimensional or three-dimensional dataset, compute PCA, and show what happens when the data are represented by fewer components.

This project shows PCA as a practical linear algebra tool for visualization and compression.

Required content

- Generate a small correlated dataset as a NumPy array.
- Center the data by subtracting feature means.
- Compute principal components using covariance eigenvectors or SVD.
- Project the data to one or two components.
- Reconstruct the data and compute reconstruction error.
- Print explained variance ratios.

Suggested approaches to compare

- Method A: keep only the first principal component.
- Method B: keep two principal components, if your data have at least three features.
- Optional Method C: compare PCA before and after standardizing the features.

Demo focus

- Plot the original data and principal direction for a 2D example.
- Print explained variance ratios.
- Show that fewer components mean smaller representation but larger reconstruction error.

Stretch goals (optional)

- Use a small real-looking dataset with features such as height, weight, speed, and score.
- Add noise and see how PCA changes.
- Build a simple slider or loop that changes the number of kept components.

#