Hand-in Exercise 1

Deadline: February 27, 13:15

Read these instructions carefully before you start coding.

This first hand-in exercise covers material from the **first three lectures** (up to and including solving linear equations). Unless noted otherwise, you are expected to code up your own routines using the algorithms discussed in class and **cannot use special library functions** (except exp(), log(), etc.). Things like `numpy` arrays and extremely simple functions like `arange`, `linspace` or `hist` (without using advanced features) are OK. Things like matrix classes, factorials and gamma functions are not. Any type of algorithm discussed in class should of course always be coded by yourself (e.g. never use `np.interp` or `scipy.interpolate`). When in doubt, write your own, or ask us on the discussion boards.

You can use the routines you wrote for the tutorials, however, your routines must be written **by yourself** (and so **in no part** by someone else or an LLM). Codes will be checked for blatant copying (with other students as well as other sources), routines which are too similar get **zero points** at best.

For every main question you should write a **separate program**/separate code file(s). We **must** be able to run everything with **a single call to a script `run.sh`**[1], which downloads any data (data needed for an exercise **may not** be included in your code package but needs to be retrieved at runtime, see the `run.sh` example), runs your scripts and generates a PDF containing all your source code and outputs **in the following format**:

- Per main question, the code of any shared modules.
- Per sub-question, an explanation of what you did.
- Per sub-question, the code specific to it.
- Per sub-question, the output(s) along with discussion/captions.

Your code may be **handed in** however you'd like, for example by emailing a zip file or by sharing a github repository. If you organize your code in multiple folders, `run.sh` should be **in the top folder**. It's a good idea to have a fellow student **test** that your code works by running ./run.sh to make sure there are no permission errors or the like. Exercises that are not run with this single command or do not have their code and output in the PDF get **zero points** (this includes solutions in Jupyter notebooks).

Ensure (**test!**) that your code runs to completion on a Virtual Desktop machine (normal workstation)[2], using the **default** version and packages of `python3` (e.g. don't load modules only installed for you). Your code should have a total run time of **at most 10 minutes**, solutions generated will not be checked beyond this limit. All scripts should run to the end without user interference, so do not ask for input and **do not use plt.show()**, as it stops automatic execution.

For all routines you write, **explain** how they work in the comments of your code and **argue** your choices! Similarly, whenever your code outputs something **clearly indicate** next to the output/in the PDF what is being printed. This includes **discussing your plots in their captions**. If you used any sources outside of this course for writing your code, **give these sources explicitly**.

If a part of your code **does not run**, explain what you did so far and what the problem was and still include that part of the code in the PDF for possible partial credit. If you are unable to get a routine to work but you need it for a follow-up question, use a library routine in the follow-up (and clearly indicate that and why you do so).

Writing code that is very time- or memory-inefficient may incur a small **penalty** (e.g. using two matrices for an LU decomposition). Additional investigations/relevant plots may earn a small **bonus** (e.g. consistency/convergence checks).

Each sub-question starts with a reference to a relevant tutorial (T) and/or a reference to the relevant lecture (L). For example, *[L2,T2.3]* means the question is related to lecture 2 and question 3 in tutorial 2. Your previously written code for these will be a great starting point!

[1]Download **and adapt** the example here: `https://github.com/daalen/NURSolutionTemplate`.
[2]For more info, see `https://helpdesk.strw.leidenuniv.nl/wiki/doku.php?id=manuals:virtualdesktopserver`.

1. **Poisson distribution**
We would like a function that returns the Poisson probability distribution for integer k:

$$P_\lambda(k) = \frac{\lambda^k e^{-\lambda}}{k!}, \tag{1}$$

given a positive mean λ. Note that the probability distribution is normalized, i.e. $\sum_{k=0}^{\infty} P_\lambda(k) = 1$ for any non-zero λ.

   (a) (4 points) *[L1,T1+2.1]* Write a function that returns $P_\lambda(k)$. When your code is run, output $P_\lambda(k)$ to at least six significant digits for these values: $(\lambda, k) = (1, 0)$, $(5, 10)$, $(3, 21)$, $(2.6, 40)$, $(100, 5)$ and $(101, 200)$ (without hard-coding). Your algorithm should avoid under/overflow well in general, not just for these specific cases.
   Important: For this exercise, we are valuing low memory usage over efficiency. That means not using standard Python types, because these can use an almost unlimited number of bytes. Instead, you need to use *only* `numpy` types of (at most) 32 bits, like `numpy.int32` or `numpy.float32`, at every step.
   Hint: If useful, you can turn this into a continuous function by taking a floating point k and rounding it. Regardless, you should find $P_\lambda(k) > 0$ for any non-zero λ and finite k – if you find zero or a negative number, think about how you could change the calculation. You don't need to use a gamma-function, but if you want to, you'll have to write it yourself!

2. **Vandermonde matrix**
In an $N \times N$ Vandermonde matrix $\mathbf{V}$, each entry is of the form $V_{ij} = x_i^j$, with $i = 0, \ldots, N-1$ the row index and $j = 0, \ldots, N-1$ the column index. It's tightly related to the Lagrange polynomial: if these x_i values correspond to the x-coordinates of a set of N points (x_i, y_i), then solving $\mathbf{Vc} = \mathbf{y}$ for $\mathbf{c}$ gives us the coefficients of the unique polynomial that passes through all points:

$$y_i = \sum_{j=0}^{N-1} c_j x_i^j \qquad \text{(for all } i = 0, \ldots, N-1) \tag{2}$$

For this exercise, use the 20 points provided at `https://home.strw.leidenuniv.nl/~daalen/Handin_files/Vandermonde.txt`.[3] An optional, minimal routine to get you started is provided at `https://home.strw.leidenuniv.nl/~daalen/Handin_files/vandermonde.py`.

   (a) (8 points) *[L3,T3.1]* Write a code that does an LU decomposition of the Vandermonde matrix for the x_i provided and solve for $\mathbf{c}$. Print out the values of $\mathbf{c}$ and plot the full 19th-degree polynomial (evaluated at ~ 1000 equally-spaced points) along with the points to show that it goes through them. Also plot the absolute difference between the given points y_i and your result $y(x)$, i.e. $|y(x) - y_i|$ (make sure the y scale is such that a difference can be seen!).

   (b) (5 points) *[L2,T1+2.3]* We know that this polynomial must be equal to the Lagrange polynomial (see lecture 2), since it too goes through all points (x_i, y_i) and is unique. Write a code that applies Neville's algorithm to all 20 points to find ~ 1000 interpolated values for the full Lagrange polynomial and plot these as well. Again, also plot the absolute difference with the given points. If the result does not lie on top of your result for (a), explain why!

   (c) (3 points) *[L3]* Let's now try to improve our result for (a) by iterating on the solution. Plot a new result (along with the absolute difference, as before) for both 1 LU iteration and 10 LU iterations and compare all results.[4] Be aware that depending on the implementation you chose for the LU decomposition, you may already be at the lowest possible error, and will therefore not see any improvement – in that case, you will still get full points for a correct implementation of the iterative improvements.

   (d) (4 points) *[L1,T1+2.2]* Finally, use `timeit` to time the execution times of (a), (b) and (c) (with 10 iterations), setting `timeit`'s `number` parameter to some appropriate number of times so we get accurate run time estimates without taking more than a minute to execute. Try to explain the difference in execution times – which is most efficient, and why? Which do you expect to yield the most accurate result, and why?

[3]Note that in a realistic scenario, you would never apply interpolation using 20 points at once! However, using this many points allows us to more easily test both efficiency and accuracy.

[4]Note: 1 iteration means one application of the error-cancelling algorithm discussed in class, not zero!