Exercises for Week 4

---

1. Broadcasting

1. **Autolab** Write a Python function `standardize_rows` which receives three inputs: `data` with shape $n \times d$, `mean` with shape d, `std` with shape d. It must then subtract `mean` from each row of `data` and then divide by `std`. Finally, it must return the result. Do not use any NumPy functions.
   Input: Three arrays: `data` with shape $n \times d$, `mean` with shape d, `std` with shape d.
   Output: a new array `output` where `output[i] = (data[i] - mean)/ std`.
   Example: For inputs
   $$\texttt{data} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \texttt{mean} = \begin{bmatrix} 0.5 & 1 & 3 \end{bmatrix} \text{ and } \texttt{std} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$
   the output should be
   $$\begin{bmatrix} 0.5 & 0.5 & 0 \\ 3.5 & 2 & 1 \end{bmatrix}.$$

2. **Autolab** Write a Python function `outer` that receives two vectors as input and returns the outer product of the two. Do not use any NumPy functions.
   Input: Two vectors as NumPy arrays of lenght n and m respectively.
   Output: The $n \times m$ matrix giving the outer product.
   Example: The outer product of $x = [1, 2]$ and $y = [3, 4, 5]$ is
   $$\begin{bmatrix} 3 & 4 & 5 \\ 6 & 8 & 10 \end{bmatrix}.$$

3. **Autolab** Write a Python function `distmat_1d` that receives two vectors as input and returns the distance matrix between them. Do not use for loops or any NumPy functions except for `abs`.
   Input: Two vectors x and y of length n and m respectively.
   Output: An $n \times m$ matrix D where $D_{ij} = |x_i - y_j|$.
   Example: For inputs $x = [1, 2]$ and $y = [3, 0.5, 1]$ the output should be
   $$\begin{bmatrix} 2 & 0.5 & 0 \\ 1 & 1.5 & 1 \end{bmatrix}.$$

2. High Performance Haversine

In this exercise, we will expand on the example from section 2.2 in Fast Python about profiling code for the Haversine distance. Consider the following Python program:

```python
import sys
import numpy as np

def distance_matrix(p1, p2):
    p1, p2 = np.radians(p1), np.radians(p2)

    D = np.empty((len(p1), len(p2)))
    for i in range(len(p1)):
        for j in range(len(p2)):
            dsin2 = np.sin(0.5 * (p1[i] - p2[j])) ** 2
            cosprod = np.cos(p1[i, 0]) * np.cos(p2[j, 0])
            a = dsin2[0] + cosprod * dsin2[1]
            D[i, j] = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))

    D *= 6371  # Earth radius in km
    return D


def load_points(fname):
    data = np.loadtxt(fname, delimiter=',', skiprows=1, usecols=(1, 2))
    return data


def distance_stats(D):
    # Extract upper triangular part to avoid duplicate entries
    assert D.shape[0] == D.shape[1], 'D must be square'
    idx = np.triu_indices(D.shape[0], k=1)
    distances = D[idx]
    return {
        'mean': float(distances.mean()),
        'std': float(distances.std()),
        'max': float(distances.max()),
        'min': float(distances.min()),
    }


fname = sys.argv[1]
points = load_points(fname)
D = distance_matrix(points, points)
stats = distance_stats(D)
print(stats)
```

This program loads a set of latitude and longitude coordinates, computes a distance matrix using the Haversine distance, and finally prints some summary statistics.

The goal of this exercise is to optimize the program through the use of profiling and NumPy. For testing we use the same weather station data as in Fast Python. We have prepared smaller subsets for fast testing. You can find all the data and subsets at `/dtu/projects/02613_2025/data/locations/`.

1. **Autolab** Make a job script that runs the above program on the hpc queue. Request a single core and specify a CPU model so the results are repeatable. For the Autolab submission, assume that the input is always a file with path `input.csv`.
2. **Autolab** Modify your job script so it runs the program under the builtin `cProfile` profiler. What do you see? In what functions does the program spend the most time?\ Hint: See section 2.1.2 and 2.2.1 in Fast Python.\ Hint: Search for the name of your Python script in the profiler output to focus on the relevant functions.
3. **Autolab** Let us optimize the `distance\_matrix` function. Using NumPy array operations and broadcasting, rewrite the function so there is only one loop instead of two nested ones.
4. Rerun the profiler. What has changed? Did it get faster? How much / little?
5. Let us now zoom in on the `distance_matrix` function using line profiling. Using the `line_profiler`, modify your Python program and job script to run with line profiling enabled on `distance_matrix`. Inspect the results. What do you see? On what lines should we focus? Hint: See section 2.2.2 in Fast Python.
6. Optimize `distance_matrix` to improve its performance. There are at least two optimizations you can make:
   a) Moving a repeated calculation out of the loop to pre-compute once.
   b) The fact that $\arctantwo\left(\sqrt{a}, \sqrt{1-a}\right) = \arcsin\left(\sqrt{a}\right)$. Re-run the profiler. Did it make the code faster? Did it change where time is spent in `distance_matrix`?
7. **Autolab** Finally, let us see what happens if we eliminate for loops completely. Rewrite `distance\_matrix` to perform all calculations with NumPy array operations and broadcasting. What happened? Did it get faster?
8. Measure the performance in MFLOP/S for the optimized single loop and the no-loop version of `distance_matrix`. Use random points as input and vary the number of points from 10^1 to 10^4. Plot the results and explain what you observe.
   a) Plot the run time in a loglog plot as a function of the size of the distance matrix in kilobytes. Include CPU cache sizes in your plot. Is one always faster? If yes, which one? If not, when do they switch?