Assignment 1

We are given a 2-dimensional grid with points $(i, j), i, j = 0, \ldots, N + 1$. In this assignment we want to simulate a discrete diffusion process on the grid. We are starting with a distribution $u_0(i, j)$ of function values on the grid points. The distribution process follows the following recurrence relation:

$$u_{n+1}(i,j) = \frac{1}{4}\left[u_n(i+1,j) + u_n(i-1,j) + u_n(i,j+1) + u_n(i,j-1)\right],\ i,j = 1,\ldots,N$$

In other words, we are simply taking the average of the neighbouring grid points. We still need to fix the boundary values. Here, we just use the condition that the boundary values should remain constant, that is

$$u_n(0,j) = u_0(0,j), \quad u_n(i,0) = u_0(i,0), \quad u_n(N+1,j) = u_0(N+1,j), \quad u_n(i,N+1) = u_0(i,N+$$

Part 1 (basic Python)

We define the following skeleton of a Python function:

```python
def diffusion_iteration(un):
    """
    Perform one diffusion step for all given grid points.

    Parameters
    ----------
    un : numpy.ndarray
        Numpy array of type `float64` and dimension (N + 2, N + 2) that stores the
        function values at step n.

    This function returns a Numpy array of dimension (N + 2, N + 2) of type `float64`
    that contains the function values after performing one step of the above diffusion
    iteration.
    """
    pass
```

Implement the function `diffusion_iteration` using pure Python without Numba acceleration. Benchmark the runtime of this function for growing dimensions N and plot the runtime against N. What overall complexity of the runtime with respect to the parameter N do you expect?

Part 2 (Numba acceleration and parallelisation)

Now optimise the function `diffusion_iteration` using Numba. In the first step develop a serial Numba implementation that does not use parallelisation. Repeat the benchmarking from the first part and compare the Numba compiled function against the pure Python version. What speed-up do you achieve with Numba? Once you have done this parallelise the function using `numba.prange`. Explain your parallelisation strategy and benchmark the resulting function. The function should parallelise almost perfectly. The optimal speed-up is roughly given by the number of physical CPU cores that you have. What is the actual speed-up that you measure compared to the theoretical speed-up?

Part 3 (Visualisation)

Assume we have some kind of material distribution u_0. Furthermore, we assume that all boundary values are 0. We now want to visualize the diffusion process by generating a nice animation. We assume the grid size parameter N to be large enough such that by the discrete time n when diffusion process arrives at the boundary, the function values u_n are negligibly small. Think about a nice initial distribution u_0 of values. Create a nice animation of 5 to 10 seconds in length that plots the iterates u_n one after another. In order to do this you can use the matplotlib function `imshow` to draw individual frames and the `FuncAnimation` class in Matplotlib to generate the animation. Some details about creating such a matplotlib animation is discussed in a Stackoverflow thread.

Advanced Problem

We could make the diffusion process more complicated by defining an index set $S = \{(i_0, j_0), (i_1, j_1), \ldots\}$ of interior indices at which we are keeping the interior iteration values constant, that is we set $u_n(i, j) = u_0(i, j)$ for $(i, j) \in S$. Implement a parallel Numba accelerated diffusion step that implements this additional condition and again create a nice visualisation. You need to change the interface of your `diffusion_interface` function to take an additional parameter `constant_indices` where you can pass the information about which indices should be kept constant. Explain the data structure you choose for this condition and how you implement it.

Assessment of the coursework

- The submission deadline for the coursework is **Monday 19 October, 10am**.
- Up to 80% of the coursework can be achieved by a perfect solution for Parts 1 to 3. The Advanced Problem is worth an additional 20%.
- The assignment does not require much code writing. But a strong emphasis is put on good explanations. Putting a few comment lines in your code is not sufficient as explanation. Use the Jupyter notebook capabilities and write good explanations about what you are doing and why you are doing it as markdown cells.
- Your code must be executable without any errors from scratch in Jupyter by choosing "Restart kernel and run all cells." If this produces any errors, any code and explanations after the error occurs will be ignored for the marking. It is not the task of the markers to debug your code.
- The code should not run for much longer than 2 minutes on a typical laptop/desktop. This is a soft limit. If your notebook runs too long we reserve the right to reject it.
- In addition to core Python packages you are allowed to use Numpy, Numba, and matplotlib. No other packages are allowed and any such request will be rejected.
- Any matplotlib output must appear inside your notebook. For more information on how to do this see here: https://medium.com/@1522933668924/using-matplotlib-in-jupyter-notebooks-comparing-methods-and-some-tips-python-c38e85b40ba1. A tutorial on embedding animations is shown here: http://louistiao.me/posts/notebooks/embedding-matplotlib-animations-in-jupyter-as-interactive-javascript-widgets/
- You must submit your solution as a single Jupyter Notebook file with ending `*.ipynb`. **Any other submission will lead to 0 marks automatically. Make sure you submit a correct Notebook file with the right ending.**.