# Week 7 - Visualization and documentation

In this exercise session you will visualize the state of the dynamical system you have implemented during the last 2 weeks with the Python package `matplotlib`. For both projects the objective is to plot relevant graphs representing quantities related to the evolution of the dynamical system and to produce short videos of the time-dependent solution. As you have seen last week, functions are useful to organize your code and to be able to reuse specific parts of it in the future. Continue using them when you write the code for your plots!

**Make sure to release your code by 10am next Monday (release notes)!**

## 1 Hopfield network

### 1.1 Energy function

A Hopfield network has an associated *energy function*. It has the form

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} p_i p_j \tag{1}$$

This expression is called *energy* because the patterns which are memorized, either with the Hebbian or with the Storkey rule, are local minima of this function. Furthermore, the energy is a non-increasing quantity of the dynamical system, meaning that the energy at time step $t$ is always greater or equal than the energy at any subsequent step $t' > t$. We want to demonstrate this property graphically.

Your task:

- Write a function `energy(state, weights)` which, given a weight matrix and a network state, it returns the energy value associated to that pattern.
- Create 50 random patterns of (network) size 2500. Perturb one of the memorized patterns by swapping the values of 1000 of its elements (from -1 to 1 or vice versa). You should execute these operations using the functions defined in previous weeks.
- Store the random patterns in a Hopfield network and make the dynamical system evolve for a maximum of 20 iterations or until convergence, first with the Hebbian weights and then with the Storkey weights, with a synchronous update rule. Use the perturbed pattern as the initial state. Store the state of the network at each time step.
- Run the dynamical system for a maximum of 30000 iterations or until convergence (we consider that convergence is reached when the state does not change for 10000 consecutive iterations), first with the Hebbian weights and then with the Storkey weights, this time with the asynchronous update rule. Store the state of the network at each time step.
- Evaluate the energy function in each of the states. Verify that the function is always non-increasing, both for the Hebbian weights and for the Storkey weights, and for both update rules, by generating a time-energy plot. Hint: computing the energy function for all states might take some time. If it takes too long, you can compute it only every 100 or even 1000 states and then plot it.

## 1.2 Visualization of the pattern evolution

We have chosen a pattern size of 2500 so that the patterns can conveniently be visualized as $50 \times 50$ pixel images. In this part, we will create a video of the evolution of the network state and verify that it converges to one of the patterns which had been initially memorized.

Your task:

- For the video, we want to reconstruct a pattern that is easily recognizable. Therefore, manually define a pattern which looks like a checkerboard:

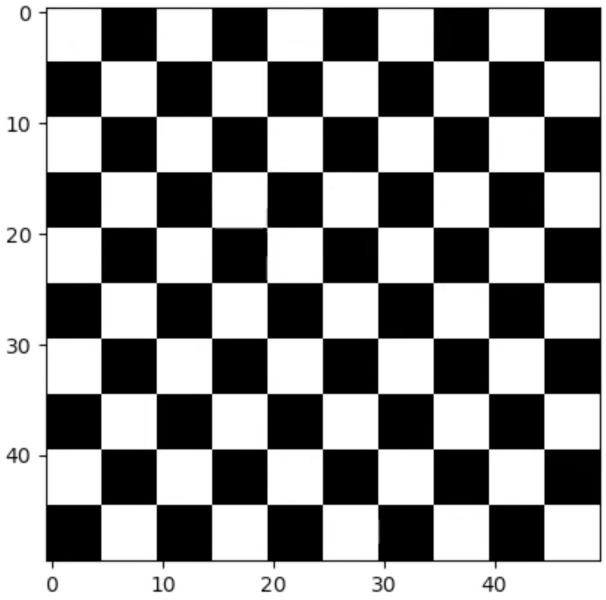


Figure 1: A 50x50 checkerboard with 5x5 checkers. This means that each checker contains 25 values, which are either 1 (white checker) or -1 (black checker).

  The elements of this pattern should be 1 where the checkerboard is white and -1 where it is black. This translates into a 50x50 matrix, in which 5x5 sub-matrices of -1 or 1 are alternated. Flatten this matrix and store it in your pattern matrix, replacing one of the random patterns.

- Generate once again the Hebbian weights based on the patterns that also include the *checkerboard pattern*.
- Perturb the *checkerboard pattern* by modifying 1000 of its elements.
- Run the dynamical system with the Hebbian weights, both with the synchronous and the asynchronous update rule. Use the same maximum number of iterations and convergence criterion as before. For the asynchronous update rule, remember not to store all the states, but only one every 1000.
- Reshape the stored states into $50 \times 50$ matrices.
- Write the function `save_video(state_list, out_path)`, which receives as input a list of states (already with the correct 50x50 shape) and generates a video of the evolution of the system. Use the function `imshow()` of the module `matplotlib.pyplot` to turn the states into

a list of frames. (Hint: to check whether the images make sense, you can use the function `matplotlib.pyplot.show()` after calling `imshow()` to visualize some of the frames). Initialize an `ArtistAnimation` from the module `matplotlib.animation` to turn the frames into a video and save it at the path `out_path`.

- Use the function `save_video()` to save a video of each of the experiments (synchronous and asynchronous execution). Verify that the network state evolves towards the checkerboard: the network remembers the original pattern and can remove the random perturbation from it!

# 2 Turing pattern generation

## 2.1 Properties of the diffusion operator

This first part focuses on the properties of the diffusion operator, which you have defined during the 5th exercise session. For the moment, we will ignore the reaction term of the equation (luckily last week you have organized your code in functions, which will make it much easier to write the code to solve a diffusion equation!). The diffusion operator can be thought as an *averaging* operator. Therefore, it has two nice properties: it preserves the total *mass* of the system (depending on the problem e.g., heat, morphogen) and it never concentrates material (at a particular location). Therefore, the sum of the solution vector is constant, while the maximum value is non-increasing.

Your task:

- Define an initial condition for the diffusion problem in which all the mass is concentrated in one point. We choose a domain of size $M = N = 101$ and we set the initial condition to 0 everywhere, apart from the central element, which we set to 1.
- Run the time-dependent simulation for 100 iterations. We choose the same parameters as in week 5: $\Delta x = 0.1$, $\Delta t = 0.0001$. Save the state at each time step.
- Plot the graph of the maximum value of the solution at each time step. Add labels to the axes and a title.
- Plot the graph of the sum of the solution at each time step. Add labels to the axes and a title.

Do the graphs show the expected behavior?

### 2.1.1 Visualize the pattern formation

From now on we move back to the full reaction-diffusion equation. The objective of this part is to create a video to show how patterns are generated from an initial random distribution of $v$ and $u$. Your task:

- Consider the reaction-diffusion problem defined during week 6. We use the same parameters: $M = 100$, $N = 100$, $\Delta x = 0.1$, $\Delta t = 0.0001$, $\alpha = 1.5$, $K = 0.125$, $\rho = 13$, $a = 103$, $b = 77$, $d = 7$, $\gamma = 0.5$. We also start again from an initial state which is a random perturbation of the stable state $(u, v) = (23, 24)$. Run a simulation for 1000 steps, while storing the time evolution of $u$ and $v$ every 10 steps (result: two lists of 100 vectors of size $100 \times 100 = 10000$ elements each, representing the time evolution of $u$ and $v$).
- Write a function `binarize(state)`, which receives a state as an input, and returns an array of values in $\{0, 1\}$, with the same shape as the input state. The values are set to $0$ if they are smaller than the average of the vector, while they are set to $1$ if they are larger.
- Convert the lists of $u^{(t)}$ and $v^{(t)}$ into binary matrices.

- Write the function `save_video(state_list, out_path)` which receives as input a list of states (a list of 100x100 Numpy arrays) and generates a video of the evolution of the system. Use the function `imshow()` of the module `matplotlib.pyplot` to turn the states into a list of frames. (Hint: to check whether the images make sense, you can use the function `matplotlib.pyplot.show()` after calling `imshow()` to visualize some of the frames). Initialize an `ArtistAnimation` from the module `matplotlib.animation` to turn the frames into a video and save it in the path `out_path`
- Use the function `save_video()` to save a video of the evolution of both $u$ and $v$. Do you see any pattern evolving from the noisy initial state?

You are now able to visualize the pattern formation process! You can now try to run the simulation for longer, even until convergence (which might take from a few seconds to some hours, depending on your computer and on the efficiency of your implementation), by changing the number of iterations.

## 3 Documentation

Documenting the code is fundamental when working in a team (and also alone!). One core component of the documentation are the comments which are written after a class definition (we will learn more about this) or a function.

In Python, there is a standard documentation template. This standard format has the advantage of being familiar for all python developers, while correctly being interpreted by software to save the documentation in .html format (e.g., Sphynx). Quick documentation lookup while using a function or a class is a standard feature of most IDEs and text editors (PyCharm, VS Code, ...), or you can access it with the command `help(<function>)`. As an example, here is part of the docstring of the function `norm` of the library `Numpy`.

```
@array_function_dispatch(_norm_dispatcher)
def norm(x, ord=None, axis=None, keepdims=False):
    """
    Matrix or vector norm.
    This function is able to return one of eight different matrix norms,
    or one of an infinite number of vector norms (described below), depending
    on the value of the ``ord`` parameter.
    Parameters
    ----------
    x : array_like
        Input array.  If `axis` is None, `x` must be 1-D or 2-D, unless `ord`
        is None. If both `axis` and `ord` are None, the 2-norm of
        ``x.ravel`` will be returned.
    ord : {non-zero int, inf, -inf, 'fro', 'nuc'}, optional
        Order of the norm (see table under ``Notes``). inf means numpy's
        `inf` object. The default is None.
    axis : {None, int, 2-tuple of ints}, optional.
        If `axis` is an integer, it specifies the axis of `x` along which to
        compute the vector norms.  If `axis` is a 2-tuple, it specifies the
        axes that hold 2-D matrices, and the matrix norms of these matrices
        are computed.  If `axis` is None then either a vector norm (when `x`
        is 1-D) or a matrix norm (when `x` is 2-D) is returned. The default
        is None.
        .. versionadded:: 1.8.0
```

```
    keepdims : bool, optional
        If this is set to True, the axes which are normed over are left in the
        result as dimensions with size one.  With this option the result will
        broadcast correctly against the original `x`.
        .. versionadded:: 1.10.0
    Returns
    -------
    n : float or ndarray
        Norm of the matrix or vector(s).
    """
```

Your task: write docstrings for **all the functions** in your code, including a short description of the behavior, of the input parameters and of the return values.

We recommend the autodocstring extension from VS code. To use it, simply install it via the extensions panel at the left or the following url: https://marketplace.visualstudio.com/items?itemName=njpwerner.autodocstring. Then, in VS code settings search for: "@ext:njpwerner.autodocstring" and set the docstring format to "numpy". This extension autogenerates a template of the docstring based on the function declaration, and then you will only need to document the inputs and outputs.