BIO-210
A. Mathis

Week 11: Object oriented programming

As you have learnt during the lectures, Python is an object-oriented language. Objects and classes are the building blocks of many Python applications. An object, as compared to a simple function, allows you to store several methods as well as state variables, which can modify the behavior of the methods. In fact, the object's methods use their input parameters, as well as the object's member variables (accessed through the *self* keyword). Furthermore, classes are an effective way to encapsulate functionalities and make them independent from the rest of the code, such that they can be reused in different parts of the code or even different projects. In this exercise section you will refactor the code that you have written during the past weeks following the principles of object-oriented programming.

Be sure to use your testing suite to not introduce bugs during refactoring. Ideally adapt the tests first and then refactor your code.

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

1 Hopfield network

Your task: implement the classes `HopfieldNetwork` and `DataSaver`, which implement the following interfaces and functionalities:

```
class HopfieldNetwork:
    def __init__(self, patterns, rule="hebbian"):
        raise NotImplementedError()

    def hebbian_weights(self, patterns):
        raise NotImplementedError()

    def storkey_weights(self, patterns):
        raise NotImplementedError()

    def update(self, state):
        raise NotImplementedError()

    def update_async(self, state):
        raise NotImplementedError()

    def dynamics(self, state, saver, max_iter=20):
        raise NotImplementedError()

    def dynamics_async(self, state, saver, max_iter=1000, convergence_num_iter=100, skip=10):
        raise NotImplementedError()

class DataSaver:
    def __init__(self):
        raise NotImplementedError()

    def reset(self):
        raise NotImplementedError()

    def store_iter(self, state, weights):
```

```
        raise NotImplementedError()

    def compute_energy(self, state, weights):
        raise NotImplementedError()

    def get_data(self):
        raise NotImplementedError

    def save_video(self, out_path, img_shape):
        raise NotImplementedError()

    def plot_energy(self):
        raise NotImplementedError()
```

The `HopfieldNetwork` class is initialized by passing the patterns and the learning rule as a string ("hebbian" or "storkey") and it generates the weights accordingly. The methods `update` and `update_async` execute one evolution step, with the synchronous and the asynchronous update rule, respectively. The methods `dynamics` and `dynamics_async` make the system evolve from an initial pattern until a certain criterion (convergence or maximum number of iterations) is met. They do not return a list of states like the corresponding functions did, but rather save the state history in a `DataSaver` object. Use the `skip` parameter to save only one every *skip* states.

The `DataSaver` class has to store the sequence of states and their associated energy in an attribute. The data can be retrieved at the end of the evolution through `get_data`, while the method `save_video` generates and stores a video of the images associated to the states. `plot_energy` generates a plot of the evolution of the energy function.

For all other functions you implemented in the last weeks, think about how to best include them in an object-oriented way, e.g. by designing new classes or extending `HopfieldNetwork` and `DataSaver`.

2 Turing pattern formation

Your task: implement the classes `TuringPattern` and `DataSaver`, which implement the following interfaces and functionalities:

```
class TuringPattern:
    def __init__(self,
                 n_x,
                 n_y,
                 d_x=0.1,
                 d_t=0.0001,
                 alpha=1.5,
                 k=0.125,
                 rho=13,
                 a=103,
                 b=77,
                 d=7,
                 gamma=300):
        raise NotImplementedError()

    def diffusion(u):
        raise NotImplementedError()
```

```python
    def update(self, u, v):
        raise NotImplementedError()

    def h(self, u, v):
        raise NotImplementedError()

    def f(self, u, v):
        raise NotImplementedError()

    def g(self, u, v):
        raise NotImplementedError()

    def diffusion_dynamics(self, u_0, saver, num_iter):
        raise NotImplementedError()

    def dynamics(self, u_0, v_0, saver, num_iter):
        raise NotImplementedError()

class DataSaver:
    def __init__(self, interval=1):
        raise NotImplementedError()

    def reset(self):
        raise NotImplementedError()

    def store_iter(self, u, v):
        raise NotImplementedError()

    def get_data(self):
        raise NotImplementedError()

    def save_video(self, out_path):
        raise NotImplementedError()
```

The `TuringPattern` class receives all the problem parameters as inputs at initialization time, which it then uses for all the other methods. During the initialization, it also assembles and stores the diffusion matrix. The method `diffusion_dynamics` executes one update of the vector **u_0**. The method `dynamics` starts from the initial values of **u** and **v** (perturbations of the stable state) and then evolves for a fixed number of iterations.

The partial values of **u** and **v** are stored by an object of the class `DataSaver`. Every *interval* iterations (the steps might be many) the object stores the partial solutions. It also implements the methods `get_data` to retrieve the data and `save_video` to store the video of the values of **u** after the threshold is applied.

For all other functions you implemented in the last classes, think about how to best include them in an object-oriented way, e.g. by designing new classes or extending `TuringPattern` and `DataSaver`.