Week 8 - Testing

Testing your code is essential to verify that it correctly behaves as you meant it to.

Creating tests early in the development of a project usually saves you a lot of time looking for hidden bugs once things get more complicated – we strongly recommend this!

This is particularly important when you work with others, but also when you use/refactor your code from a long time ago. Testing makes sure that any changes to the code, are still producing the relevant results (for previous features).

This week we will focus on *unit testing*, a form of automated testing aimed at checking the behavior of individual parts of your code in isolation (i.e., units). We will use the python libraries `doctest` and pytest to write tests.

doctest

You can test functions using doctest[1]. For a specific example to understand how doctest works, you can have a look at the doctests feature developed in the demo-project (https://github.com/EPFL-BIO-210/demo-project/tree/feature/doctests). For example, you can look at the function `dX_dt` in the `LotkaVolterraModel.py`:

```
def dX_dt(X, a=1.0, b=0.1, c=1.5, d=0.75):
    """
    Computes the growth rate of fox and rabbit populations based on system state (X)
    and parameters (a,b,c,d)
    Parameters
    ----------
    X : array or tuple
        [prey_count, predator_count]
    a : float, optional
        natural growth rate of the prey (rabbit)
    b : float, optional
        natural dying rate of the prey
    c : float, optional
        natural growth rate of the predator (fox)
    d : float, optional
        natural dying rate of the predator
    Returns
    -------
    numpy array
        [change of prey_count, change of predator_count]

    Examples
    -------
    >>> dX_dt(np.ones(2),1,0.1,1.5,.75)
    array([ 0.9  , -1.425])
    >>> dX_dt(np.zeros(2),1,0.1,1.5,.75)        # zero is a fixpoint
    array([0., 0.])

    """
    return np.array([a * X[0] - b * X[0] * X[1], -c * X[1] + d * b * X[0] * X[1]])
```

[1] Source: https://docs.python.org/3/library/doctest.html

Here, when doctest runs, it parses the docstrings and tests the function after the symbols ">>>". The subsequent row defines he expected output.

You can then test the functions in the `LotkaVolterraModel` module by writing on the terminal `python LotkaVolterraModel.py`. It should have no output (if all tests pass). You can also print a report running by running `python LotkaVolterraModel.py -v`.

Doctests, are fantastic and very easy to implement, especially for simple function calls. To test more complex scenarios, `pytest` is more appropriate!

pytest

One core concept of unit testing is the *assertion*. Writing an assertion is like asking the code to verify that a certain condition (equality, inequality, truth, ...) is verified. We can have a look at the following example from the `pytest` docs[2], where the function `func` is tested:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

You can then run the example by writing on the terminal `pytest test_sample.py`. You should observe a failure report as `func(3)` does not return 5.
Remark: Test functions must begin with the prefix `test_`.
Remark: The scripts that contain your test function(s) have to begin with `test_` or end with `_test`.

If you want to test many functions, you can add multiple tests (as functions) in a single script (as a reference looks at the `test_LVM.py`: https://github.com/EPFL-BIO-210/demo-project/blob/feature/doctests/test_LVM.py). For (unit-tests), each test should focus on one aspect of the function to test, so we expect to have just one or a few assertions per test. In the example, you can observe that the previous doctests can be integrated with pytest using `assert doctest.testmod(LotkaVolterraModel, raise_on_error=True)`. This is useful to run your test-suite with a single call.

You can run the demo-project tests by writing on the terminal `pytest test_LVM.py`. Tests can both verify regular behavior (`assert`) or exceptional behavior (`pytest.raises(Exception)`.

Line coverage

A widely used metric to evaluate how extensively your code is tested is *line coverage*, which counts the lines of code of your project which are executed at least once by a test.

Importantly, even with 100% line coverage you cannot be sure that your code is bug-free. It is up to you to design tests which perform important checks in your code, at least guaranteeing that it behaves as expected in those particular circumstances. Some python IDEs, such as PyCharm, integrate a testing interface, which computes many metrics (including the line coverage) automatically. Otherwise, you can use the coverage module[3]. For instance, together with the developed pytest you can get the coverage by running in the terminal `coverage run -m pytest test_LVM.py` to collect the results. To see the coverage report, simply type `coverage report -m`.

This week your task is to develop unit tests and to try to get as close as possible to 100% line coverage. We list some reasonable tests for the two projects below, but you can design many more.

[2] Source: https://docs.pytest.org/en/6.2.x/getting-started.html#create-your-first-test
[3] Source: https://coverage.readthedocs.io

1 Hopfield network

- Test that the weights matrix with the Hebbian weights has the correct size, that its values are in the correct range, that it is symmetric, that the elements on the diagonal are 0
- Consider the following patterns:

$$\begin{pmatrix}1 & 1 & -1 & -1\end{pmatrix} \quad \begin{pmatrix}1 & 1 & -1 & 1\end{pmatrix} \quad \begin{pmatrix}-1 & 1 & -1 & 1\end{pmatrix}$$

create a test to check that the Hebbian weights are (for doctests, be careful at using the correct number of digits)

$$\begin{pmatrix} 0. & 0.33333333 & -0.33333333 & -0.33333333 \\ 0.33333333 & 0. & -1. & 0.33333333 \\ -0.33333333 & -1. & 0. & -0.33333333 \\ -0.33333333 & 0.33333333 & -0.33333333 & 0. \end{pmatrix} \tag{1}$$

while the Storkey weights are:

$$\begin{pmatrix} 1.125 & 0. & -0.1875 & -0.5625 \\ 0.5 & 0.625 & -0.6875 & 0.1875 \\ -0.1875 & -0.8125 & 0.75 & 0. \\ 0.0625 & 0.1875 & -0.5 & 1. \end{pmatrix}$$

Hint: the function `allclose` of the library `numpy` might be useful to compare matrices of floats.

- Test the evolution of the system, for example, by checking that it converges to the correct pattern and that the energy function is non-increasing
- Devise additional tests to reach as high line coverage as possible. You can also improve your code to deal with violations of the requirements of the input (e.g., raise an exception if the patterns are non-binary or if they are not unique)

2 Turing pattern formation

- Create tests that check the value of the functions f, g and h.

  Use the following parameters: $u = 1$, $v = 2$, $\alpha = 1.5$, $K = 0.125$, $\rho = 13$, $a = 103$, $b = 77$,

  the corresponding values must be: $h = 12.235294117647058$, $f = 89.76470588235294$, $g = 100.26470588235294$
- Test that the function `diffusion` applied to the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

returns the following matrix:

$$\begin{pmatrix} 2 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & -1 & -1 & -2 \end{pmatrix}$$

- Use the following parameters: $M = 3$, $N = 3$, $\Delta x = 0.1$, $\Delta t = 0.0001$, $\alpha = 1.5$, $K = 0.125$, $\rho = 13$, $a = 103$, $b = 77$, $d = 7$, $\gamma = 0.5$ and start from the initial states

$$u_0 = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \qquad v_0 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

  Check that the values of u and v after one iteration are:

$$u_1 = \begin{pmatrix} 0.04515 & 1.03448824 & 2.02393571 \\ 3.01347805 & 4.00309286 & 4.99276301 \\ 5.98247609 & 6.97222301 & 7.96199706 \end{pmatrix} \qquad v_1 = \begin{pmatrix} 1.2857 & 2.21501324 & 3.14443571 \\ 4.07395305 & 5.00354286 & 5.93318801 \\ 6.86287609 & 7.79259801 & 8.72234706 \end{pmatrix}$$

  , while after 5 iterations are:

$$u_5 = \begin{pmatrix} 0.22108069 & 1.1686221 & 2.11711012 \\ 3.0661668 & 4.01550672 & 4.96509061 \\ 5.91485142 & 6.86476954 & 7.81479791 \end{pmatrix} \qquad v_5 = \begin{pmatrix} 2.24356641 & 2.9363427 & 3.62979046 \\ 4.32391978 & 5.01802638 & 5.7123312 \\ 6.40676441 & 7.10139079 & 7.79610334 \end{pmatrix}$$

- Test the evolution of the system, for example, by checking that the state has always the correct size, or that if you remove the reaction terms the maximum value is non-increasing and the sum is constant.

  Hint: the function `allclose` of the library `numpy` might be useful to check if the sum is constant.

- Devise additional tests to reach as high line coverage as possible. You can also improve your code to deal with violations of the requirements of the input (e.g., raise an exception if the patterns are non-binary or if they are not unique)