Unit test and integration test

The aim of this exercise is to create a test suite for the the Jacobi solver. In particular, you will create unit test for the functions setBoundaryConditions and update method, as well as an integration test.

The stratey for C and python will be slightly different, so will provide separate indications

C

For this exercise we provide a stub of run_test.c and a Makefile

Unit test

update method

Let's first write a unit test for the update method. What we want to check is that the stencil calculation is correct, on a small example.

To this end, we use a 3x3 grid and fill it with 1 only in the elements that will be used by the update routine, namely

```
|0 1 0|
|1 0 1|
|0 1 0|
```

Given the update formula, this should yield 1 on the center of the buffer.

The run_test.c contains the stub of the function, some indications about the steps to take and the main function to run all the tests.

The steps are

- allocate the buffers
- fill them with zeros and set to 1 the aformentioned elements
- call the update function
- return the value of the central elements

The makefile contains the a "test" target the compiles and runs the testing suite

```
make test
```

You can try now to modify the update function (possibly introducing some bugs) and verify that issuing 'make test' will warn you.

setBoundaryConditions method

Next, we will write a small unit test for setBoundaryConditions. In this case we would like to verify that the same boundary conditions are set on both the grid buffers. This because the algorithm never updates the boundary, so if the buffers do not match, different iterations will use different b.c., leading to utterly wrong results.

Your task is to

- allocate the buffers
- fill them with zeros
- call the setBoundaryConditions
- verify that the buffers match, at least in the borders

Once again, the run_test.c contains the stub of the function, some indications about the steps to take and the main function to run all the tests.

To verify that the test works, in the jacobi_functions.c you can find a different set of b.c. (namely, all the borders set to 100). Comment the current boundary conditions and uncomment the new ones. Does this version pass the test?

integration test

For simplicity, we will use a run of the entire code as integration test. This is done by having a reference output (called solution_test.dat, provided, run with parameters dimension = 10, iterations = 5) and to have the 'make test' run the whole code with those paramters.

To enable it, uncomment the lines in the makefile

@echo "" @./$(TARGET) 10 5 > /dev/null @cmp -s solution_test.dat solution.dat;
RETVAL=$$?;
if [$$RETVAL -eq 0]; then
echo " Integration test: passed!";
else
echo " Integration test: failed!"; \

Clearly, if one of the unit test fails, the integration tests fails. Moreover, if there are issues about the integration of the different functions, this should be spotted by this last step. For example, use the folloing b.c. in the setBoundaryConditions function

```
for ( i = 0; i <= dimension + 1; i++){
    grid[ i * ( dimension ) ] = 100.0;
    grid[ i * ( dimension ) + dimension + 1] = 100;
    grid[ i ] = 100;
    grid[ ( ( dimension + 1 ) * ( dimension ) ) + i ] = 100;
    gridNew[ i * ( dimension ) ] = 100.0;
    gridNew[ i * ( dimension ) + dimension + 1] = 100;
    gridNew[ i ] = 100;
    gridNew[ ( ( dimension + 1 ) * ( dimension ) ) + i ] = 100;
}
```

Verify that those b.c. pass the unit test, but fails the integration test (and fix it!)

Python

For this exercise we provide a stub of run_test.py. This example uses the built-in module unittest. Alle the tests will be member functions of a subclass of unittest.TestCase

Unit test

update method

Let's first write a unit test for the update method. What we want to check is that the stencil calculation is correct, on a small example.

To this end, we use a 3x3 grid and fill it with 1 only in the elements that will be used by the update routine, namely

```
|0 1 0|
|1 0 1|
|0 1 0|
```

Given the update formula, this should yield 1 on the center of the buffer.

The run_test.c contains the stub of the function, some indications about the steps to take and the main function to run all the tests.

The steps are

- allocate the buffers
- fill them with zeros and set to 1 the aformentioned elements
- call the update function
- return the value of the central elements

To run the suite

```
python run_test.py
```

You can try now to modify the update function (possibly introducing some bugs) and verify that issuing 'python run_test.py' will warn you.

setBoundaryConditions method

Next, we will write a small unit test for setBoundaryConditions. In this case we would like to verify that the same boundary conditions are set on both the grid buffers. This because the algorithm never updates the boundary, so if the buffers do not match, different iterations will use different b.c., leading to utterly wrong results.

Your task is to

- allocate the buffers
- fill them with zeros
- call the setBoundaryConditions
- verify that the buffers match, at least in the borders

Once again, the run_test.py contains the stub of the function, some indications about the steps to take and the main function to run all the tests.

To verify that the test works, in the jacobi_functions.py you can find a different set of b.c. (namely, all the borders set to 100). Comment the current boundary conditions and uncomment the new ones. Does this version pass the test?

integration test

The integration test in this case is done inside the test suite, by calling a minimalistic version of the full code. You can find it in the 'test_integration' method. The logic is as in the C case, that is running all the functions on a predetermined system size and compare it with a given output file (called solution_test.dat, provided, run with parameters dimension = 10, iterations = 5)

Clearly, if one of the unit test fails, the integration tests fails. Moreover, if there are issues about the integration of the different functions, this should be spotted by this last step. For example, use the folloing b.c. in the setBoundaryConditions function

```
for i in range(dimension + 2):
    grid[i][1]  = 100.0;
    grid[i][-1] = 100;
    grid[1][i]  = 100;
    grid[-1][i] = 100;

    gridNew[i][1]  = 100.0;
    gridNew[i][-1] = 100;
    gridNew[1][i]  = 100;
    gridNew[-1][i] = 100;
```

Verify that those b.c. pass the unit test, but fails the integration test (and fix it!)