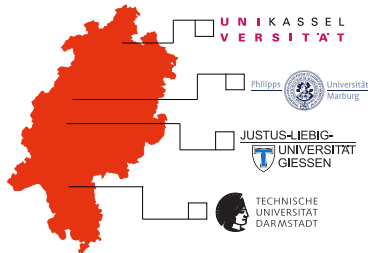


Testing of scientific software

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

Tim Jammer, Contributions by Dr. Marcel Giar

October 17, 2022



HKHLR is funded by the Hessian Ministry of Sciences and Arts



In this talk:

- ▶ Why do we need to test scientific software?
- ▶ Different testing methods
- ▶ How to write good tests?
- ▶ Problems of testing scientific software in particular
- ▶ Numerical considerations
- ▶ Using test frameworks

- ▶ Testing builds confidence that your code is indeed correct

- ▶ Testing builds confidence that your code is indeed correct
- ▶ If you are not confident that the code is correct: All your scientific results derived from that are basically worthless
 - ▶ Same way as when an experiment is done where you are not sure if it has been conducted correctly according to the current state-of-the-art methodologies
- ▶ Worst case: you need to **retract a publication** if the used code turned out to be faulty Examples: <https://doi.org/10.1126/science.314.5807.1856>
- ▶ Software is often used in the scientific process but the software itself is often not developed according to the scientific standard
- ▶ Nowadays more often: Publishing code alongside publication is mandatory

- ▶ In computer science one can distinguish different testing strategies:
 - ▶ Unit-Testing
 - ▶ Integration Testing
 - ▶ System Testing
 - ▶ Regression Testing

- ▶ **Basic idea is the same**

- ▶ In computer science one can distinguish different testing strategies:
 - ▶ Unit-Testing
 - ▶ Integration Testing
 - ▶ System Testing
 - ▶ Regression Testing

- ▶ **Basic idea is the same**
- ▶ Testing: compare what *should be* the result to the *actual* result

- ▶ **Existence**
 - ▶ Correctness
 - ▶ Completeness (coverage)
 - ▶ Readability/understandability
 - ▶ Easy to run
 - ▶ Fast to run
 - ▶ Deterministic
 - ▶ Reliable
-
- ▶ Not all characteristics always achievable

- ▶ Testing methods are unknown
- ▶ No time/budget for testing
- ▶ Testing the model and not the code
- ▶ "Missing Oracle"
- ▶ Interpretability of results

- ▶ Testing methods are unknown
 - ▶ Hopefully has changed after this course
- ▶ No time/budget for testing
- ▶ Testing the model and not the code
- ▶ "Missing Oracle"
- ▶ Interpretability of results

- ▶ Testing methods are unknown
- ▶ No time/budget for testing
 - ▶ As software is often also published, more focus is put on it
 - ▶ Will hopefully rise the awareness that testing is indeed important
 - ▶ Nevertheless testing can save development time
- ▶ Testing the model and not the code
- ▶ "Missing Oracle"
- ▶ Interpretability of results

- ▶ Testing methods are unknown
- ▶ No time/budget for testing
- ▶ Testing the model and not the code
 - ▶ We want to test if the code actually reflects the model you build
 - ▶ Testing if the model actually resembles reality is part of the scientific work
 - ▶ If a code does not resemble the model but produces "correct" results the model is false
- ▶ "Missing Oracle"
- ▶ Interpretability of results

- ▶ Testing methods are unknown
- ▶ No time/budget for testing
- ▶ Testing the model and not the code
- ▶ "Missing Oracle"
 - ▶ Science is exploratory by its nature
 - ▶ So what are the correct results for our new software?
 - ▶ Some techniques to deal with this problem are covered later in this course
- ▶ Interpretability of results

- ▶ Testing methods are unknown
- ▶ No time/budget for testing
- ▶ Testing the model and not the code
- ▶ "Missing Oracle"
- ▶ Interpretability of results
 - ▶ Computers have rounding errors
 - ▶ Also some simplifications can be part of the model, leading to further inaccuracies
 - ▶ Is a "wrong" result an acceptable rounding error or a fault?
 - ▶ Next slides

- ▶ Computers can not represent infinite accuracy (machine epsilon)
- ▶ Computers use binary representation
 - ▶ $\frac{1}{10} = 0.1$ in decimal but $0.000110011\overline{0011}$ in binary
 - ▶ Same reasoning as why $\frac{1}{6} = 0.1\overline{6}$ in decimal

- ▶ Computers can not represent infinite accuracy (machine epsilon)
- ▶ Computers use binary representation
 - ▶ $\frac{1}{10} = 0.1$ in decimal but $0.000110011\overline{0011}$ in binary
 - ▶ Same reasoning as why $\frac{1}{6} = 0.1\overline{6}$ in decimal
- ▶ Some algorithms are only approximations
 - ▶ Sometimes accuracy - runtime trade-off

- ▶ Computers can not represent infinite accuracy (machine epsilon)
- ▶ Computers use binary representation
 - ▶ $\frac{1}{10} = 0.1$ in decimal but $0.00011001\overline{10011}$ in binary
 - ▶ Same reasoning as why $\frac{1}{6} = 0.1\overline{6}$ in decimal
- ▶ Some algorithms are only approximations
 - ▶ Sometimes accuracy - runtime trade-off
- ▶ Floating point arithmetic has rounding errors!
 - ▶ Examples with 32 bit IEEE floating point representation:
 - ▶ $(0.001 + 1) - 1 = 0.00099999999999998899$
 - ▶ but $(1 - 1) + 0.001 = 0.001$
 - ▶ $1 * 10^{10} + 1 = 1 * 10^{10}$

⇒ Compare floating point number with error margins in mind

▶ These machine rounding errors can behave differently if

- ▶ Degree of concurrency (race conditions)
- ▶ Using a different parallelization scheme
- ▶ Using different compiler option
- ▶ Using a different architecture
- ▶ ...

⇒ Test your code again when circumstances of execution change!

▶ Reproducible vs Replicable

- ▶ Tools can help you analyzing floating point arithmetic related issues in your code:
<http://fpanalysistools.org/>

- ▶ Test against real experiment data
- ▶ Test against an analytical solution
- ▶ Test against other implementations
- ▶ Metamorphic testing
 - ▶ When I double value X , i know that this should affect the result in a specified way
- ▶ Regression testing
 - ▶ When I change some implementation detail, the core functionality should be the same
- ▶ Implicit faults
 - ▶ Segmentation-fault, data races, divide by zero, ... are always errors

- ▶ Testing frameworks can ease running all the testcases in an automated fashion
- ▶ But they are not required to have a well tested application
- ▶ Example: pytest for python


```
1 import pytest
2
3 def m(x):
4     return 2 * x
5
6 def d(x):
7     return 2 / x
8
9
10
11 def test_function_2():
12     assert m(2) == 4, "assertion message correct"
13
14 @pytest.mark.parametrize("test_input,expected", [(4, 8), (16, 32), (64, 128)])
15 def test_function_x(test_input, expected):
16     assert m(test_input) == expected
17
18 def test_function_by_zero():
19     # tell Pytest to expect a ZeroDivisionError:
```


- ▶ Functions named with `test_` prefix are considered tests
- ▶ `assert` statements are used to signal test result
- ▶ Functions can be parameterized
- ▶ Run with `pytest simple_example.py`
- ▶ More information: <https://docs.pytest.org/en/7.1.x/how-to/index.html>

- ▶ One can also generate input data
- ▶ For Python, one can use the hypothesis package, which integrates with pytest
- ▶ More information:
<https://hypothesis.readthedocs.io/en/latest/quickstart.html>
- ▶ More complex structures can also be generated e.g.
`hypothesis.extra.numpy.arrays`
(<https://hypothesis.readthedocs.io/en/latest/numpy.html>)

Examples/simple_example.py

```
1      d(2)
2
3  from hypothesis import given, settings, strategies as st
4
5  @settings(max_examples=1000000000000000)
6  @given(st.integers(min_value=0,max_value=1000))
```


- ▶ You will write some tests for testing the power method for finding the largest eigenvalue of a matrix.
 - ▶ https://en.wikipedia.org/wiki/Power_iteration
- ▶ It is an iterative approach that refines the solution, until a satisfactory accuracy is reached
- ▶ The code for this method and some initial tests are given
- ▶ A good introduction about eigenvalues and eigenvectors can be found here:
 - ▶ <https://www.youtube.com/watch?v=PFDu9oVAE-g>

- ▶ For each of the existing testcases:
Which of the presented strategies for getting a Test Oracle do they employ?
- ▶ For each presented strategy to get the Test Oracle:
Can you come up with some testcases?
- ▶ There is at least one case in which the procedure does not work correctly.
Can you find an example input for this case?
- ▶ Can you fix the procedure so that this case also works correctly?
- ▶ Do regression testing! After your change, do all other testcases still work correctly?
- ▶ Play around with the numerical accuracy by changing the tolerance and/or number of iterations. Do you get different results? Do you need to adjust the testcases?
- ▶ You can also try to use `np.float32` as the datatype. Do you see different results?

- ▶ Test your software!
- ▶ Testing:
Compare *actual* result against the *expected* result
- ▶ Some methods for dealing with the Missing Oracle problem
- ▶ At least do regression testing
- ▶ If you fix a bug, add a test to make sure that bug will never occur again

- ▶ Computers can only generate pseudo random numbers
 - ▶ Pseudo random number generators are initialized with a seed
 - ▶ The "random" numbers you get depend on the seed
- ⇒ The same seed will lead to the same sequence of random numbers
- ▶ For replicating a specific behaviour one may need to use the same random seed
 - ▶ The seed can be set by using `np.random.seed()` or `random.seed()` (depending on the pseudo random number generator in use)
-
- ▶ One can also randomize the order of the test execution with the `pytest-random-order` module, which enables the `--random-order` parameter

- ▶ Code coverage measures how many lines of the tested code are executed by the tests
- ⇒ If a line is never executed by a test, it is never tested
- ▶ A high coverage builds confidence in the tests completeness
- ▶ 100% coverage does **not** mean that the tests are indeed complete

- ▶ Python has the coverage package.
- ▶ For integration with pytest, I recommend the `pytest-cov` plugin.
 - ▶ Run Tests and collect coverage data:
`pytest --cov=power_method test_power_method.py`
 - ▶ A file `.coverage` will be created
 - ▶ `coverage report` can be used in order to view the coverage report
 - ▶ `coverage html` produces a html report annotating which lines are covered by the tests


```
1 name: test-power-method
2 on:
3   push
4
5
6 jobs:
7   run-tests:
8     runs-on: ubuntu-latest
9     steps:
10      - uses: actions/checkout@v3
11      # prerequisites
12      - run : pip install -r requirements.txt
13      # python itself is already installed in the ubuntu image
14      # run the tests
15      - name: Run Tests
16        run: |
17          pytest test_power_method.py
```

- One can further automate running the tests using a CI pipeline



Example: <https://github.com/ttm02/ci-demo>