



Technische  
Universität  
Braunschweig



Suresoft workshops series  
**Introduction to Software Testing**

Harikrishnan Sreekumar and Yannik Hüpel, 8th August 2022

# Workshop objectives

- Familiarize with testing concepts from a **research software perspective**
- How to **incorporate testing** in our code development routines
- Capable of **coding** basic unit tests, acceptance tests and code quality checks

# Agenda

Introduction to Suresoft

Motivation for testing research software

Fundamentals of software testing

Major types of software testing

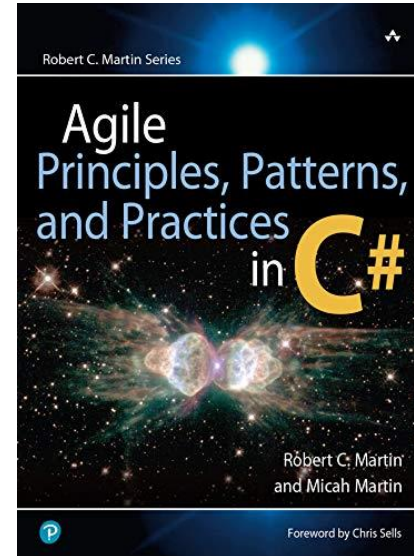
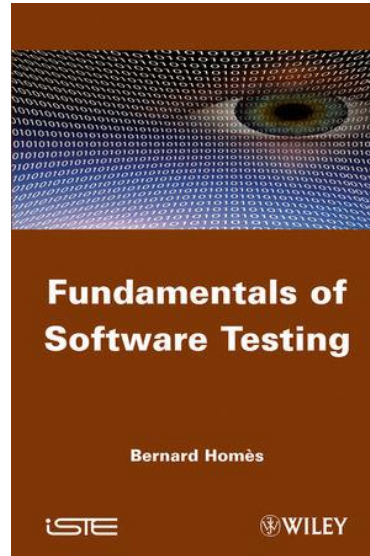
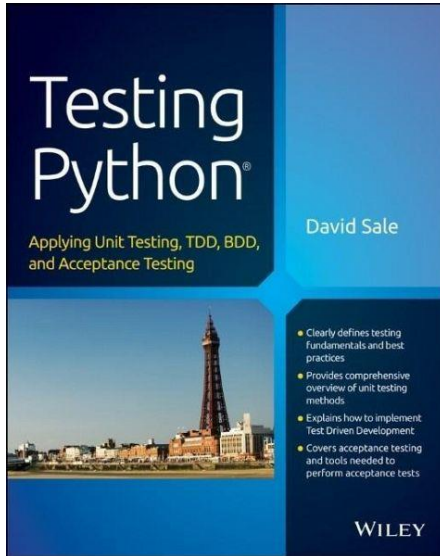
- Unit tests and code coverage + Hands-on with PyTest + Break
- Acceptance tests + Hands-on with PyTest + Short Break
- Code quality tests + Demonstration (with PyLint)
- Test process automation + Demonstration (with Paver) + Short Break

Demonstration of the in-house code eIPaSo's test environment

# Information

- Workshop slides and documentation (more details, commands, hints, ...)   
<https://suresoft.gitlab-pages.rz.tu-bs.de/workshop-website>
- We look forward to **your questions and experiences** – please unmute and interrupt anytime during the workshop or post in chat
- **Workshop preparation** – see in workshop documentation
  - Visual Studio Code
  - Python3 installation
  - Example code project
- We use the main room for our hands-on session – no break-out rooms
- We use **python** as our standard language

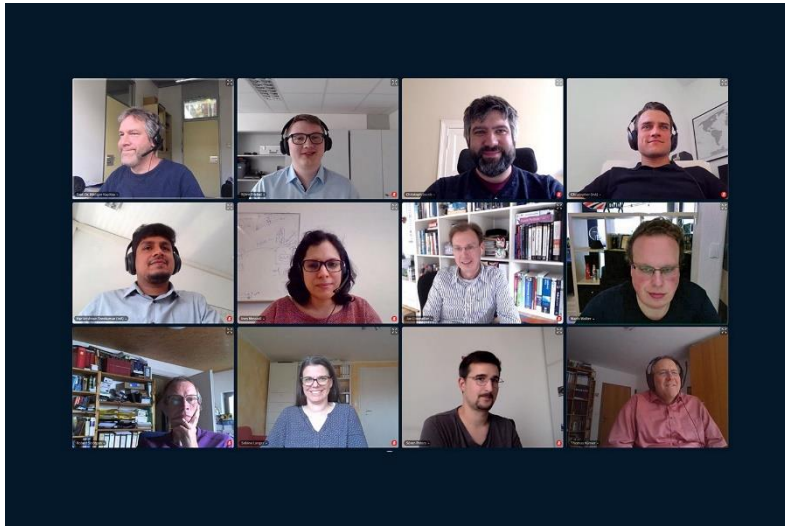
# Literature recommendation



# Introduction to Suresoft

# Who are we?

18 People from 7 Institutes and Facilities



Institut für Physikalische  
und Theoretische Chemie



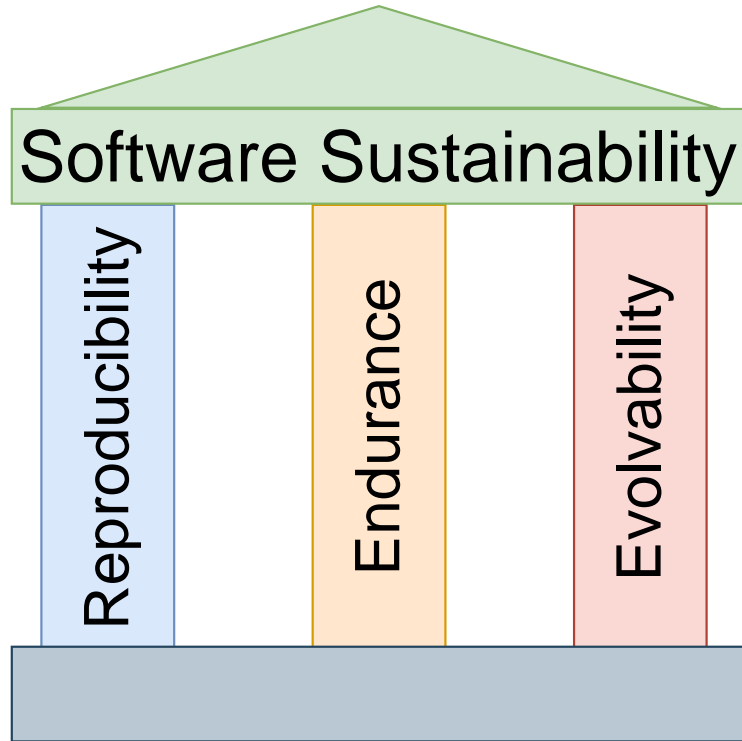
University Library &  
Gauß-IT-Zentrum

# Common problems of research software

1. Software has low code quality
2. Software is neither published nor documented
3. Software depends on a specific runtime environment (e.g third party libraries), which may not be available to other researchers



# Software sustainability



# SURESOFTE Approach for Sustainable Software

## Education

Documentation

Software Engineering  
Principles

Testing

## Infrastructure & Methods

Version Control

Archiving &  
Publication

CI &  
Automated Testing

Virtualization

Issue Reporting

Installation &  
Deployment

# Suresoft workshop series

## Every 4 weeks

- |  |                 |
|--|-----------------|
| 1. Version Control using Git   | June 13         |
| 2. Clean Code and Refactoring  | July 11         |
| <b>3. Introduction to Software Testing</b>                                       | <b>August 8</b> |
| 4. Introduction to Continuous Integration (CI) using GitLab and Containerization | September 5     |
| 5. Principles of Software Engineering  | October 3       |
| 6. Introduction to Design Patterns   | October 31      |
| 7. Working with legacy code  | November 28     |
| 8. Test Driven Development   | January 23      |
| 9. Documentation   | February 20     |

# Motivation for testing in science

# Motivation | Software testing? Why?



[<https://www.codementor.io>]

# Motivation | Causes of hidden software bug - An Example

## Ariane 5 – The Worst Software Bugs in History



Photo source: [https:// www.esa.int](https://www.esa.int)

Article: <https://www.bugsnag.com/blog/bug-day-ariane-5-disaster>

# Motivation | Necessity for software testing

- First step towards **code sustainability**
- Ensures and documents the **correct behaviour** of a software
- Contributes to the overall software **quality**
- Quickly **identifies defects/bugs** in a developing code → save time for debugging

“Scientists spend 57% of the time finding and fixing bugs”

[P. Prabhu et al., A Survey of the Practice of Computational Science, 2011]

# Motivation | Suresoft survey results

- **24.14%** do not consider testing because of lack of time
- **27.59%** add tests to old codes
- **41.38%** miss sufficient knowledge for testing



# Fundamentals of software testing

“Software testing shows the presence of bugs, not their absence”

[<https://www.hexacta.com/testing-in-software-more-than-finding-bugs/>]

# Definition of software testing

“Software testing is a set of **activities with the objective of identifying failures** in a software or system and to evaluate its level of quality.”

[B. Homès: Fundamentals of Software Testing. 2012]

“Software testing is the process of **executing a program with the intend of finding errors.**”

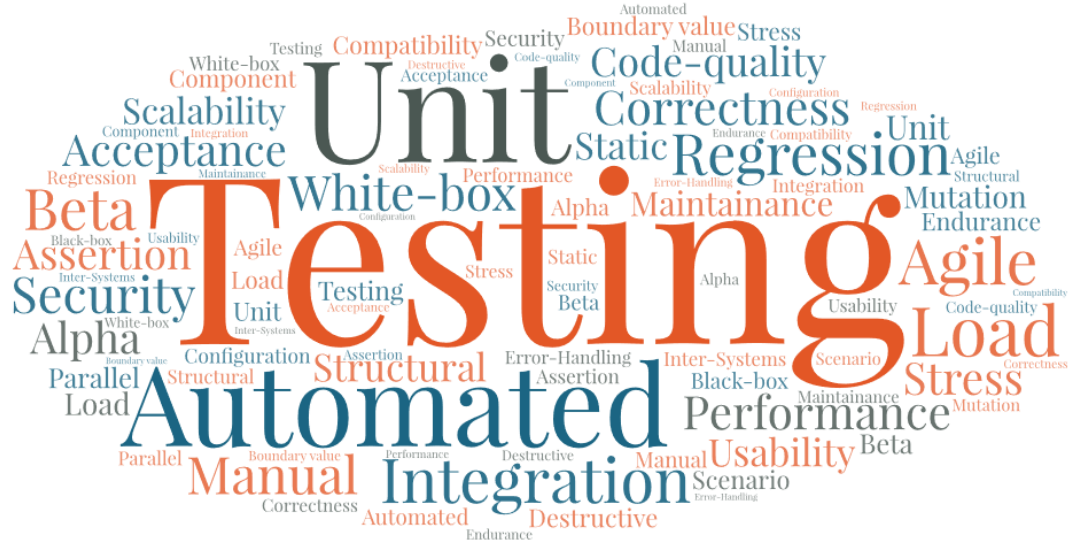
[J. M. Myers et al.: The Art of Software Testing. 2011]

“Software testing is the process of **evaluating and verifying** that a software product or application does what it is supposed to do.”

[IBM: What is software testing? <https://www.ibm.com/topics/software-testing>]

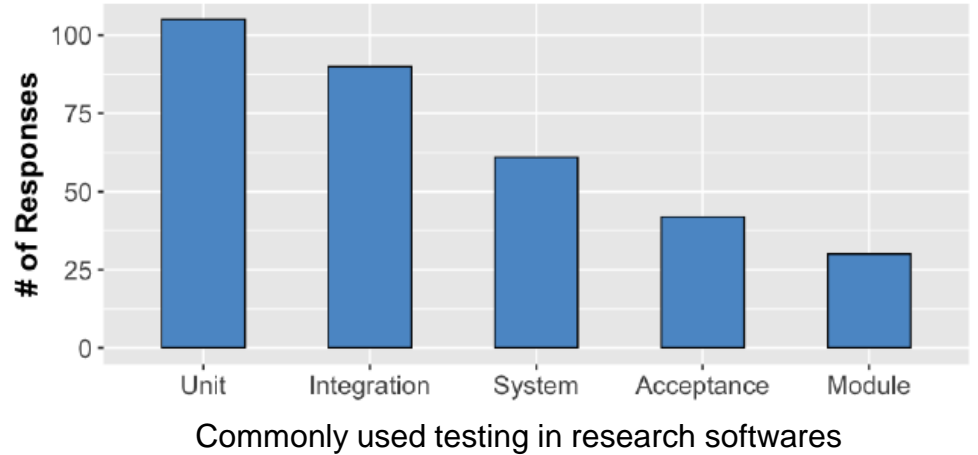
# Types of software testing

150+ types of tests and still increasing...



→ But, what are relevant for research software?

# Software testing for research softwares



[N. U. Eisty, et al.: Testing Research Software: A Survey. 2022]

Workshop focus → Unit testing, acceptance testing, code-quality testing

# Functional and non-functional tests

## Functional tests

- Focus on the **proper functioning of the software** and it's components
- Example: Correctness/accuracy (unit-, acceptance testing)

## Non-functional tests

- Focus on the **non-functional aspects** like performance, software's usability, code quality, stability, testability, adaptability, portability, etc.



# Manual and automated testing

## Manual testing

- Oldest methods
- Typically done by a QA tester (black-box)
- Tests different features of the software

## Automated testing

- Most efficient – faster and more aspects are tested
- Main component of continuous integration and deployment
- Typically done by the developer with the help of testing tools (white-box)



Focus of this workshop

# Typical automated software testing framework



## System under test (SUT)

Software itself or software components



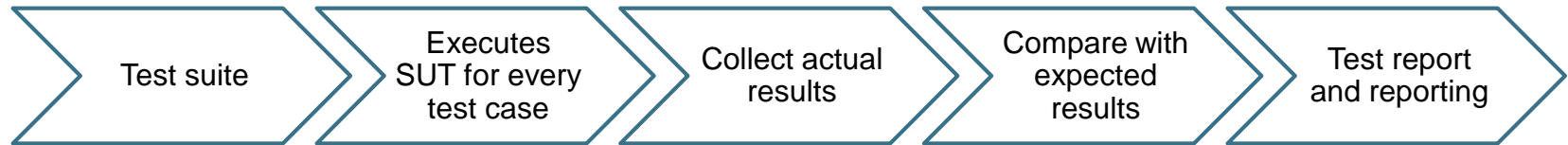
## Test suite: Test cases + Expected results

Benchmarked software's expected behaviour and test specification



## Test reports

Test status (Success/Failed) with test measures



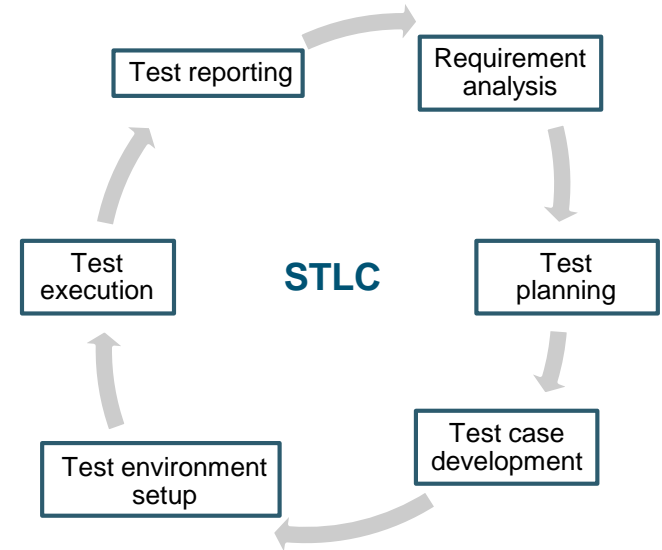
## Test harness



# How to incorporate testing in practice?

## Software testing life-cycle (STLC)

- Testing is always present in a software development cycle
- Sequential/Iterative/Incremental methodology to achieve a level of quality
- Agile model example: Test driven development (TDD) → Suresoft Workshop on TDD

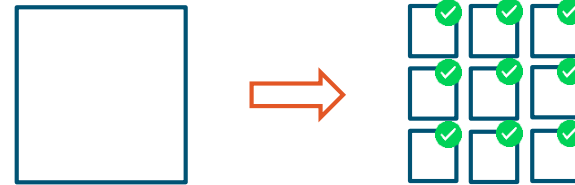


# Unit testing and Hands-on session

8 August 2022 | Harikrishnan Sreekumar, Yannik Hüpel | SURESOFT – Software Testing | Page 26

# Unit testing

- Tests a code at its basic level
- Codes are isolated - according to their specific functionalities - into smaller units and tested for proper operation

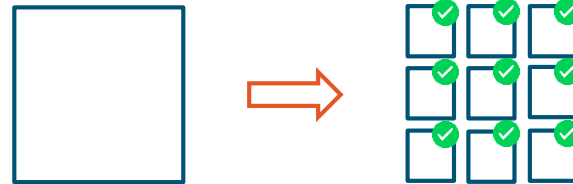


“Unit testing is more of an act of design than of verification. It is more of an act of documentation than of verification.”

[R. C. Martin and M. Martin: Agile Principles, Patterns and Practices in C#. 2006]

# How to incorporate unit-testing?

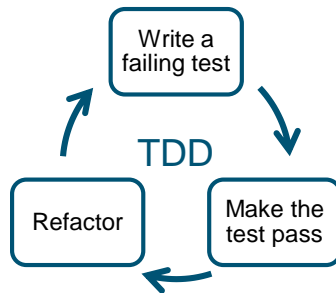
- Existing code? Breakdown into very small functions



- Writing new code? Easy! Follow an unit-testing methodology from the very start.

→ Test Driven Development (TDD)

[K. Beck: Test-Driven Development. 2002] [Suresoft TDD Workshop]



# How to write tests? The AAA Pattern

- Three A's: Arrange, Act and Assert
  - Added advantage is that the tests are **easily readable**
- 
- **Arrange** : Requirements to test the functions are prepared
  - **Act** : Function under test is called and output is collected
  - **Assert** : The expected operation of the function is checked

```
def XYZ():  
    ...  
    ...  
  
def test_XYZ():  
    # Arrange  
    _____  
    _____  
    # Act  
    _____  
    _____  
    # Assert  
    _____  
    _____
```

# Tools for unit-testing

Python

- `pytest` [<https://docs.pytest.org/en/stable/>]
- `unittest` [<https://docs.python.org/3/library/unittest.html>]

C++

- `GoogleTest` [<https://github.com/google/googletest>]

# Assertions

- Assertions **checks** whether the outcome meet certain expectations
- Boolean expression: true means assertion success and false means assertion fail

PyTest uses python's standard `assert`:

- `assert 1 == 1 # success`
- `assert "Hello" == "Hallo" # fails`
- `assert 3.14159265359 == pytest.approx(3.14, 1e-3) # success`

# Test metrics: Code coverage

- Analysis method which determines the **amount of code executed** by a test suite and which are not.
- We aim for the **best code coverage** with unit testing
- Code coverage types:
  - Functional coverage : how many functions are tested
  - Branch coverage : how many execution paths are tested
  - Line/**statement coverage** : how many lines of code/statements are tested
- Coverage tools

Python

- **coverage** [ <https://coverage.readthedocs.io/en/6.4.2/> ]

C++

- **GNU gcov + lcov** [ <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, <https://github.com/linux-test-project/lcov> ]
- **Intel codecov + profmerge**



# Hands-on | Writing your first unit test

## Example project – Matrix Calculator

- Perform basic matrix operations: **Add, Multiply, Inverse**
- Can handle different matrix format: **Dense**
- Can handle user-written linear solvers: **Jacobi iterative solver**

# Hands-on | Writing your first unit test

## Example project – Matrix Calculator

```
|- src  
  |-- MatrixAlgebra  
  |--- dense_matrix.py
```

---

### Matrix Addition

add(A, B)

$$C = A + B$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

---

### Matrix-Vector Multiply

matrix\_vector\_multiply(A, b)

$$c = A \times b$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$

---

### Matrix Inverse

matrix\_inverse(A)

$$C = A^{-1}$$

$$\begin{bmatrix} 10 & 1 \\ 3 & 8 \end{bmatrix}^{-1} = \begin{bmatrix} 0.1039 & -0.0130 \\ -0.0390 & 0.1299 \end{bmatrix}$$

# Hands-on | Writing your first unit test

## Example project – Matrix Calculator

```
|- src
  |-- MatrixAlgebra
  |--- dense_matrix.py
  |-- MatrixSolver
  |--- jacobi_solver.py
```

**How to start with unit-testing?**  
→ **Demonstration**

---

**Solve**

`solve(A, b)`

$$c = A^{-1}b$$

$$\begin{bmatrix} 10 & 1 \\ 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 0.6233 \\ 0.7662 \end{bmatrix}$$

---

`jacobi_solver` uses the `dense_matrix` functionalities

# Hands-on | Writing your first unit test

## Start testing and increase code coverage to 100% | 20 minutes

- Write 3 unit tests to test the functions `add`, `matrix_vector_multiply`, `matrix_inverse` implemented in `dense_matrix.py` | XX% CC
- (Optional) Write additional tests where matrix entries are float values and use `pytest.approx` | XX% CC

---

### Matrix Addition

`add(A, B)`

$$C = A + B$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

---

### Matrix-Vector Multiply

`matrix_vector_multiply(A, b)`

$$c = A \times b$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$$

---

### Matrix Inverse

`matrix_inverse(A)`

$$C = A^{-1}$$

$$\begin{bmatrix} 10 & 1 \\ 3 & 8 \end{bmatrix}^{-1} = \begin{bmatrix} 0.1039 & -0.0130 \\ -0.0390 & 0.1299 \end{bmatrix}$$

# Acceptance testing and Hands-on session

# Acceptance testing

- Tests the **application as a whole** and ensure proper operation
- Acceptance testing perform **verification**
- Documentation of stable application state and execution
- Black box testing



“If unit testing verifies that the code does exactly what the programmer expects it to do, then acceptance testing verifies that the code does what the user expects it to do.”

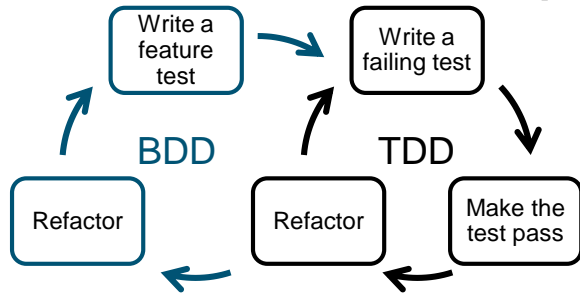
[D. Sale: Testing Python: Applying Unit Testing, TDD, BDD and Acceptance Testing. 2014]

# How to incorporate acceptance-testing?

- Design specific test cases which executes certain features of the application
- Aim for **maximum code coverage** with the various test cases
- Follow an acceptance-testing methodology in your development life-cycle

→ **Behavior Driven Development (BDD)**

[D. Sale: Testing Python: Applying Unit Testing, TDD, BDD and Acceptance Testing. 2014]



We will still use the AAA pattern!

# Tools for acceptance-testing

Python

- `pytest` [ <https://docs.pytest.org/en/stable/> ]
- `robot` [ <https://robotframework.org/> ]
- **Custom made ???**



# Hands-on | Writing your first acceptance test

## Example project – Matrix Calculator

| - main.py

**How to start with acceptance-testing?**  
→ **Demonstration**

---

### Case “add”

Performs addition of supplied matrix data

`py main.py --add`

---

### Case “solve”

Performs solving of supplied matrix data

`py main.py --solve`

---

### Case “default”

Exits with a failure code: `exit(-1)`

`py main.py xyz`

---

# Hands-on | Writing your first acceptance test

## Start testing and increase code coverage to 100% | 10 minutes

- Write an acceptance test to check the “solve” case | XX% CC
- (Optional) Write an `application-death-test` to check “default” case | XX% CC

---

### Case “solve”

Performs solving of supplied matrix data

`main('--solve')`

Assert with reference solution →  
`loadmat('./data/ref_result_system100x100_solve.mat')['result']`

---

### Case “default”

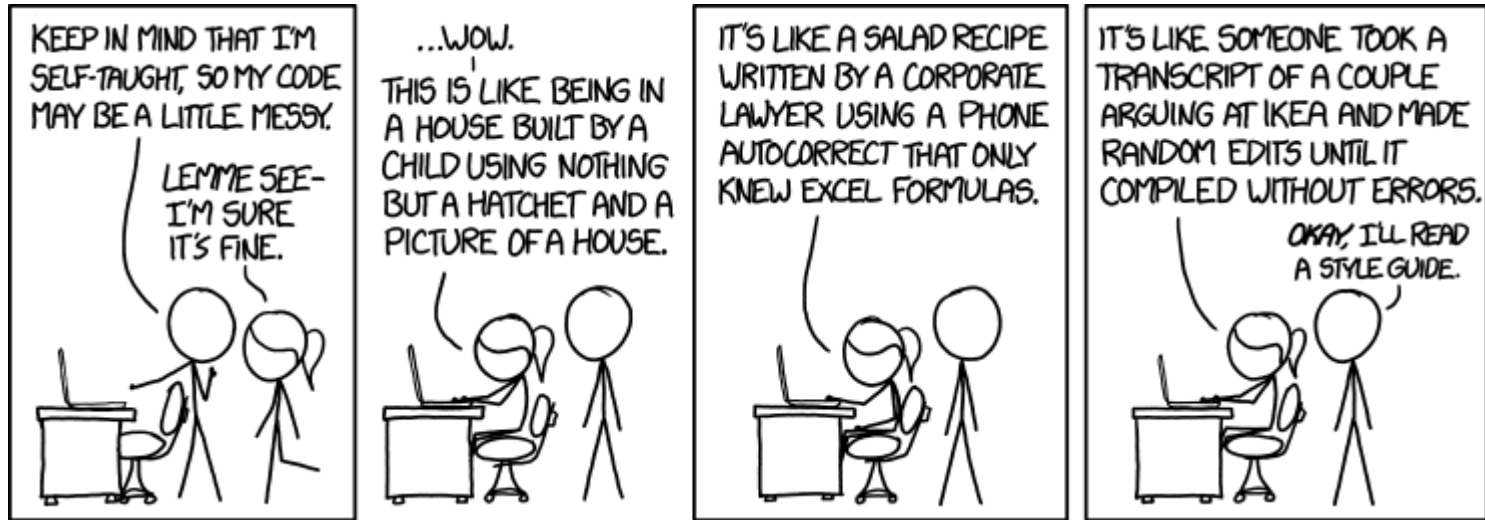
Exits with a failure code: `exit(-1)`

`main('xyz')`

---

# Code-quality testing and demonstration

# Code quality testing



[[What is code quality, how to measure and improve code quality? \(codegrip.tech\)](https://codegrip.tech)]

# Code quality testing


- Quality code consists of those features that cater to the need of customers and subsequently provide product satisfaction
- Quality code is free from deficiencies
- Quality code measures how well code can communicate between developers

# Motivation for code quality testing

- Poor quality code tends to die early because it might entail substantial technical debt
- Quality code makes your software:
  - More sustainable (minimum changes over time)
  - Robust (can cope with error usage)
  - Promotes easy transferability
  - Increases readability
  - Decreases technical debt

# How do we conduct code quality checks?

- Occurance of software defects and software quality are related
- Code quality gets overlooked in favor of programming speed → Can accumulate to a huge workload

 **Lint**er is a tool that automatically checks the quality of the code fitting to your conventions

# Tools for code-quality checks

Python	<b>PyLint</b> <b>Flake8</b>	[ <a href="https://pylint.pycqa.org/en/latest/">https://pylint.pycqa.org/en/latest/</a> ] [ <a href="https://flake8.pycqa.org/en/latest/index.html">https://flake8.pycqa.org/en/latest/index.html</a> ]
C++	<b>Clang-Tidy</b>	[ <a href="https://clang.llvm.org/extra/clang-tidy/">https://clang.llvm.org/extra/clang-tidy/</a> ]
...	<b>SonarQube</b>	[ <a href="https://www.sonarqube.org/">https://www.sonarqube.org/</a> ]



# Test process automation and demonstration

# Test process automation

Testing procedures are repetitive and time consuming

The testing process can easily be conducted by a script running automatically

→ Test process automation

What is test process automation?

- Automating the testing procedure
- Automating the management and application of test data and results

[What Is Automation Testing? \(codecademy.com\)](https://www.codecademy.com)

# Test process automation

Testing procedures are repetitive and time consuming

The testing process can easily be conducted by a script running automatically

→ Test process automation

What is test process automation?

- Automating the testing process
- Automating the management and application of test data and results

80% of organizations use automation testing and it is projected to increase in the next years

# Motivation for test automation

- Cost  
Automated testing will lead to testing without manpower
- Speed  
More tests can be concluded in the same amount of time
- Effectiveness  
Usually automated tests find bugs sooner

# What are easily automated tests?

Repetitive tests

Time-consuming tests

Tests for multiple builds

Tests vulnerable to  
human error

Frequently used tests

# How do we automate testing?



# Test process automation with Paver

Paver is one tool for automating tests in python

A `pavement.py` file with our tests has to be written



Demonstration

# Tips for software testing



# Tips for software testing

- Choose the best suitable type of testing for your code → [Start with unit-testing](#)
- Always [write tests first](#) before writing production code → Forces the system to be testable → Suresoft Workshop on “TDD”
- Designing test codes for legacy codes → Break dependencies and refactor codes to make them testable → Suresoft workshop on “Working with legacy codes”
- Design [clear and simple test cases](#)
- Defining a set of domain specific [benchmarks](#)
- Benchmarks are often [computationally expensive](#) → Connect your tests to run on a high-performance computing cluster (HPC-Rocket, Jacamar CI)
- Tests are done in specific environments → Containerization Workshop

# Demonstration of eIPaSo testing framework

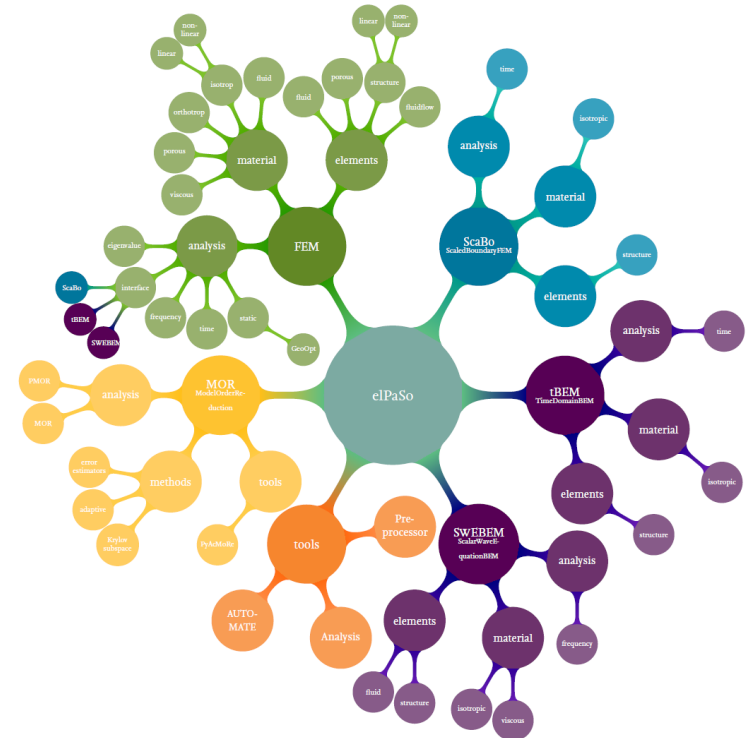
# eIPaSo | About

## Elementary Parallel Solver (eIPaSo)

- Performs vibroacoustic analysis in the modal, static, time and frequency domain
- Based on FEM, BEM, SBFEM
- Efficient computing strategies - parallel computing, model order reduction



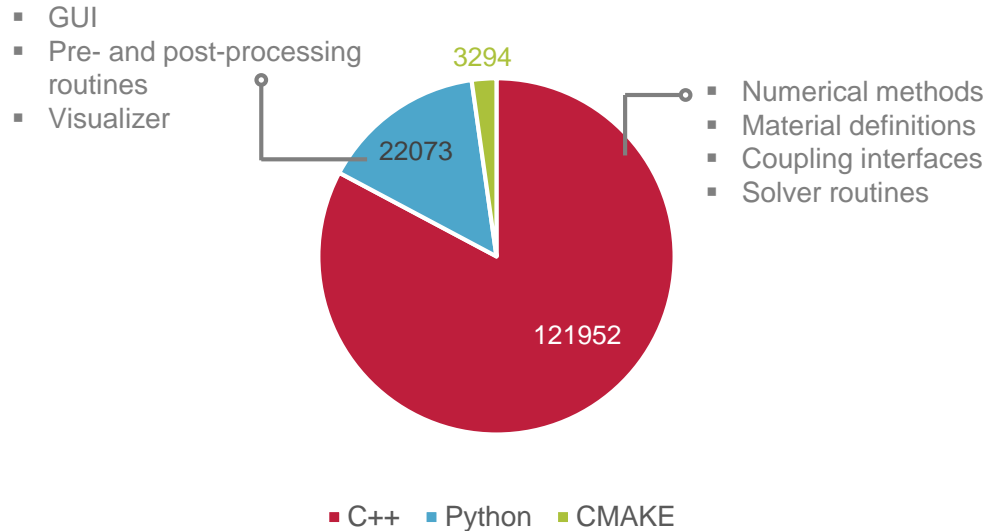
[tu-bs.de/en/ina/institute/ina-tech/research-code-elpaso](https://tu-bs.de/en/ina/institute/ina-tech/research-code-elpaso)



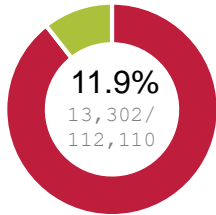
Source: InA/TU Braunschweig

# eIPaSo | Source code

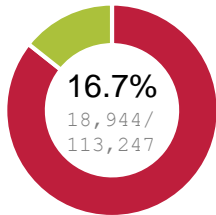
Programming language and SLOC:



# eIPaSo | Testing Framework



Unit test  
coverage



Acceptance test  
coverage

## Unit Testing

Google Test  
361 tests

## Acceptance Testing

eIPaSo AUTOMATE Tool  
36 tests

## Performance Testing

eIPaSo AUTOMATE Tool  
4 tests

## Code Quality Checks

Clang-Tidy



Vibroacoustic  
Benchmark  
Repository

- **Verification benchmarks**  
(previous eIPaSo versions or ABAQUS)
- **Validation benchmarks**  
(from experiments)
- **Performance benchmarks**  
(Scalability with MPI and OMP threads)

# eIPaSo | How tests are incorporated?

## Unit testing

- New codes → Test driven development
- Legacy codes → Refactoring and make it testable

→ Demonstration

## Acceptance testing

- New feature → New benchmark

## Features of the eIPaSo AUTOMATE Tool

- Python tool running eIPaSo benchmarks and compare with set reference
- Execute tests in Phoenix Cluster with HPC-Rocket for computationally expensive tests
- Issue reporting – `python-gitlab` for automated issue creation in GITLAB issue board
- Detailed technical report (currently generated as PDF, in future also as Gitlab pages)

# Coming up next in workshop series

# Suresoft workshop series

## Every 4 weeks

1. Version Control using Git June 13
2. Clean Code and Refactoring July 11
3. Introduction to Software Testing August 8
- 4. Introduction to Continuous Integration (CI) using GitLab and Containerization September 5**

5. Principles of Software Engineering October 9
6. Introduction to Design Patterns October 31
7. **Highlights** November 28
  - Essentials and Hands-on exercises to
    - 8. Test Drive Development
      - configure a CI pipeline in TU BS GitLab January 23
      - 9. Documentation
        - create containers using Docker and host in GitLab February 20



**Register Here:**  
[lnk.tu-bs.de/u1J3Hn](https://lnk.tu-bs.de/u1J3Hn)  
[tu-bs.de/suresoft](https://tu-bs.de/suresoft)



**Thank you for your attention**