Computing at Scale

Lecture 11: Testing

Jacob Merson

February 18, 2025

Logistics

Today's Agenda

- Inheritance (Quickly)
- Why Test?
- Unit Testing
- Integration Testing
- Testing Frameworks (Catch2)

Inheritance

Consider using Separate Inheritance Hierarchies

- Consider using separate inheritance hierarchies for different purposes.
- Use a pure virtual base class for the interface.
- Use multiple inheritance to represent combined functionality.

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#
c137-use-virtual-bases-to-avoid-overly-general-base-classes

Testing

What is Testing?

Software Testing helps

- check if your software does what you expect it to do
- helps identify bugs
- may help identify performance bottlenecks
- helps identify and validate constraints or other important design considerations

Benefits of Testing

- helps to write more moduluar and maintainable code
- increases confidence that your code is doing what's expected
- enables you to refactor your code with confidence
- catch regressions in software performance and functionality

I don’t have time to test!

A piece of advice I wish I had gotten as a graduate student: “You don’t have time not to test your code.”

Without testing, you will spend more time debugging and fixing your code than if you had written tests in the first place.

Without testing, your code will be more fragile and harder to maintain.

Without testing, you may not know if your code is providing you with the correct results.

Scenario

You are working on putting together results for a conference paper that is due in a week. You checked your code against an analytical solution by hand about a month ago. But, in a last minute push to add a feature, you make a bunch of changes to your code. And everything looks qualitatively similar. But, upon closer inspection you notice that the results are not correct.

Without tests, you are looking at a long week of stressful evenings praying you find the bug.

With tests, you can run `git bisect` and quickly identify the commit that introduced the bug. Once you fix the bug, you can rerun your tests to ensure that the bug is fixed.

Lesson's Learned

- Tests should be run often (ideally after every change).
- Tests should be inexpensive to run.
- Tests should be as granular as possible to get the most detailed view on what's failing.

Types of Testing

- Unit Testing: Testing individual functions or methods.
- Integration Testing: Testing how different parts of your code work together.
- System Testing: Testing the entire system (often lumped with integration testing).
- Acceptance Testing: Testing that the software meets the formal requirements (not typical for scientific applications to have formal requirement documents).

Unit Testing

The idea of unit testing is simple. Test the basic building blocks of your code to ensure they are working as expected.

- For each function you write check that the output is expected for the range of inputs you expect.
- For each class test the public interface. Let encapsulation do the rest.
- Don't `#define private public` to test private methods. This is a code smell.

Tips for Defining Functions

- Each function should have a clear purpose, and a single responsibility that can be described by its name.
- Functions should be small.
- Locality of behavior and data is key to maintainability.
- If you find yourself writing a comment “step 1: do operation x” this is likely a function boundary.
- If you aren’t sure what to name it, pick a horrible name and refactor later.
- If a function is hard to test, it is probably doing too much.

Integration / System Testing (Verification)

In scientific computing, integration and systems testing ensures that our code is producing the correct scientific results. This is often referred to as **verification**. I.e., is my code solving the equations correctly?

Methods:

- Compare to analytical solution
- Method of manufactured solutions
- Compare to other codes

Validation

Validation is the process of checking if you have chosen the right model to represent the real physics of the system you are studying.

Methods:

- Compare to experimental results
- Compare to other validated codes or models

Test Driven Development (TDD)

Test driven development is a process where you write the tests before you write the code.

1. Write a test (that fails)
2. Write code to make test pass
3. Refactor code and tests as needed

Testing Frameworks

- Catch2
- Google Test
- Doctest
- Boost Test
- Boost-UT

Catch2

Catch2 is a C++ testing framework that is easy to use. I find it easier to use that google test, or other commonly used testing frameworks.

Example:

```
#include <catch2/catch_test_macros.hpp>
#include <cstdint>

uint32_t factorial( uint32_t number ) {
    return number <= 1 ? number : factorial(number-1) * number;
}

TEST_CASE( "Factorials␣are␣computed", "[factorial]" ) {
    REQUIRE( factorial( 1) == 1 );
    REQUIRE( factorial( 2) == 2 );
    REQUIRE( factorial( 3) == 6 );
    REQUIRE( factorial(10) == 3'628'800 );
}
```

Integration With CMake

CmakeLists.txt

```
find_package(Catch2 3 REQUIRED)
# These tests can use the Catch2-provided main
add_executable(tests test.cpp)
target_link_libraries(tests PRIVATE Catch2::Catch2WithMain)

# These tests need their own main
add_executable(custom-main-tests test.cpp test-main.cpp)
target_link_libraries(custom-main-tests PRIVATE Catch2::Catch2)

# register tests in ctest
include(CTest)
include(Catch)
catch_discover_tests(tests)
```

https://github.com/catchorg/Catch2/blob/devel/docs/cmake-integration.md

Catch2 Terminology / Syntax

- `TEST_CASE(test name, [tags,])`: Defines a test case
- `SECTION(section name, [section description, ])`: sections are a way to group tests together than share common setup. Replace fixtures in other testing frameworks.
- `REQUIRE(expression)`: If the expression is false, the test fails.
- `REQUIRE_FALSE(expression)`: If the expression is true, the test fails.
- `CHECK(expression)`: If the expression is false, the test fails, but the test continues.
- `CHECK_FALSE(expression)`: If the expression is true, the test fails, but the test continues.
- `REQUIRE_THROWS(expression)`: If the expression does not throw an exception, the test fails.

https://github.com/catchorg/Catch2/blob/devel/docs/assertions.md

Section Example

```
TEST_CASE("vector␣resizing") {
  std::vector<double> v{3};
  REQUIRE(v.size() == 3);
  SECTION("resizing␣bigger␣changes␣size") {
    v.resize(10);
    REQUIRE(v.size() == 10);
    REQUIRE(v.capacity() >= 10);
  }
  SECTION("resizing␣smaller␣changes␣size␣but␣not␣capacity") {
    v.resize(1);
    REQUIRE(v.size() == 1);
    REQUIRE(v.capacity() >= 3);
  }
}
```

Templated Tests

```
TEMPLATE_TEST_CASE("vector␣resizing", int, double, (std::pair<int,
    int>)) {
  std::vector<double> v{3};
  REQUIRE(v.size() == 3);
  SECTION("resizing␣bigger␣changes␣size") {
    v.resize(10);
    REQUIRE(v.size() == 10);
    REQUIRE(v.capacity() >= 10);
  }
  SECTION("resizing␣smaller␣changes␣size␣but␣not␣capacity") {
    v.resize(1);
    REQUIRE(v.size() == 1);
    REQUIRE(v.capacity() >= 3);
  }
}
```

https://github.com/catchorg/Catch2/blob/devel/docs/test-cases-and-sections.md

Checking Floating Point Numbers

Remember, you should never use equality comparison of floating point numbers. Catch2 has a built in way to compare floating point numbers and vectors. (Note `REQUIRE_THAT`)

```
REQUIRE_THAT(computation(input),
             Catch::Matchers::WithinRel(expected, 0.001) ||
             Catch::Matchers::WithinAbs(0, 0.000001) );

REQUIRE_THAT(1.0, WithinRel(1.1, 0.1));
REQUIRE_THAT(1.1, WithinRel(1.1, 0.1));
```

https://github.com/catchorg/Catch2/blob/devel/docs/matchers.md
https://github.com/catchorg/Catch2/blob/devel/docs/assertions.md#floating-point-comparisons

Checking Vectors

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_vector.hpp>
#include <numeric>

using Catch::Matchers::Approx;
using Catch::Matchers::Equals;

TEST_CASE("vectors␣close"){
    std::vector<double> v1(3);
    std::vector<double> v2 = {0,0,0};
    REQUIRE_THAT(v1, Approx(v2));

    std::vector<int> v3(3);
    std::iota(v3.begin(), v3.end(), 0);
    std::vector<int> v4 = {0,0,0};
    REQUIRE_THAT(v3, Equals(v4));
}
```

Additional Bits

- “Mocking” can be used to isolate parts of your code like calls to a network.
- “Fuzz Testing” can be used to test how your code behaves under random inputs.
- Code coverage tools can show you how much of your code is being tested. Law of diminishing returns applies.
- Continious Integration (CI) will automatically run your tests when you push to your repository (more next week).

Summary

- Testing will feel *hard* at first, but it will get easier.
- Testing will save you time in the long run.
- Testing will help you define more clear abstractions and interfaces. This makes your code more moduluar and maintainable.
- Testing both helps you catch bugs earlier.
- Testing helps refactor without fear.
- Testing helps ensure that you are meeting your scientific goals.
- Testing is a critcal part of any scientific software and

In Class Exercise

Using your code from last class to compute the square root of a number using newton's method, use catch2 to write a set of basic tests.

Remember: to get the square root of a number a, you need to find the root of $f(x) = x^2 - a$.

In Class Exercise

Implement test cases for your templated vector.

In Class Exercise

Use test driven development to write a function that computes the roots of a quadratic equation. The function should take the coefficients of the quadratic equation and return the roots.