Testing and Debugging (PHY1610H lecture 11)

Ramses van Zon

February 15, 2022

1

Motivation

Three bits of reality about scientific software:

- **Scientific software can be large, complex and subtle.**
- **Scientific software is constantly evolving.**
- **Code will be handed down, shared, reused.**

Example of this complexity

Consider the sample code to simulate a damped wave equation in one dimension. It had to

1. Read parameters;
2. Set initial conditions;
3. Compute the evolution of the wave in time;
4. Output the result.

At some point in the research project, initial conditions may need to change, or the output, or the algortihm to compute the time evolution, . . .

Managing complexity using modularity

- Modularity is extracing the different parts of the program that are responsible for different things.
- Each of these should be fairly independent.
- Implementation changes of one module should not affect other modules.
- Each part can be maintained by a different person.
- Once a part is working well, it can be used as an appliance.

Questions

1. How do we ensure a module works correctly?

   $\longrightarrow$ **Unit testing**

2. What if we find that it doesn't?

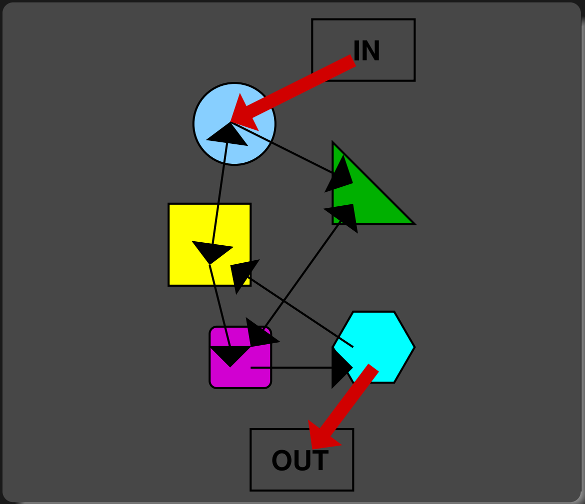
   $\longrightarrow$ **Debugging**

2

Unit testing

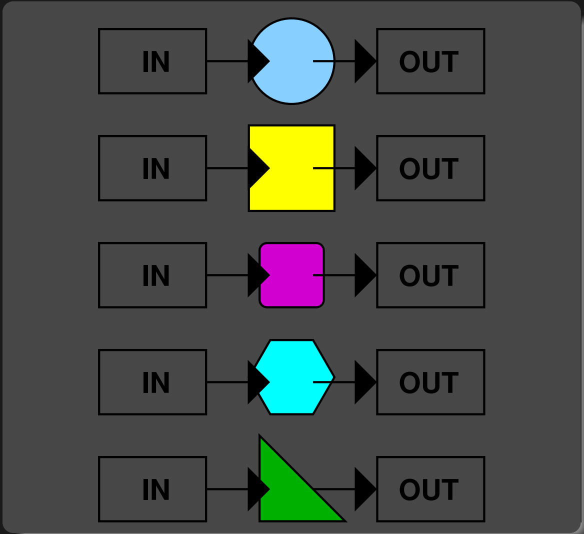
Integrated testing

- Especially with new software, or old software that was modified, you'll want to verify that it *works*.
- Test the application with a smaller test case for which you know that output.
- This can strictly only prove incorrectness (no tests can prove correctness).
- But if no errors are found, it increases your level of confidence in the software.



Unit testing

- An integrated test essentially gives you one data point.
- If you've modularized the code into *n* parts, you should have at least *n* data points to know that the parts aren't failing.
- Because each module has one responsibility, you can write a test for each module.
- If the test for a module fails, you only need to inspect that module, not the whole code of the application.
- Note that if you did not modularize, everything is connected, you could not have (n) test this. And when the integrated test fails, the error could be anywhere in the code.



Example from lecture 4 (unmodular)

```cpp
// hydrogen_monolyth.cpp
#include <iostream>
#include <fstream>
#include <rarray>
const int n = 4913;
rmatrix<double> m(n,n);
rvector<double> a(n);
double b = 0.0;
void pw() {
    rvector<double> q(n);
    q.fill(0.0);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            q[i] += m[i][j]*a[j];
    a = q;
}
double en() {
    double e = 0.0, z = 0.0;
    for (int i = 0; i < n; i++) {
        z += a[i]*a[i];
        for (int j = 0; j < n; j++) {
            e += a[i]*m[i][j]*a[j];
        }
    }
    return b + e/z;
}
```

```cpp
int main()
{
    a.fill(1.0);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            m[i][j] = H(i,j,n);
        }
    }
    for (int i = 0; i < n; i++)
        if (m[i][i] > b)
            b = m[i][i];
    for (int i = 0; i < n; i++)
        m[i][i] -= b;
    for (int p = 0; p < 20; p++)
        pw();
    std::cout<<"Ground state energy="<<en()<<"\n";
    std::ofstream f("data.txt");
    for (int i = 0; i < n; i++)
        f << a[i] << std::endl;
    std::ofstream g("data.bin", std::ios::binary);
    g.write((char*)(&a[0]), a.size()*sizeof(a[0]));
}
```

Example from lecture 4 (modular)

```cpp
// hydrogen.cpp
#include <iostream>
#include <rarray>
#include "eigenval.h"
#include "outputarr.h"
#include "initmat.h"
int main() {
    const int n = 4913;
    rmatrix<double> m;
    rvector<double> a;
    double e;
    initMatrix(n, m);
    groundState(m, e, a);
    std::cout<<"Ground state energy="<<e<<"\n";
    writeText("data.txt", a);
    writeBinary("data.bin", a);
}
```

```make
# Makefile
CXXFLAGS=-std=c++17 -O2 -g
all: hydrogen
hydrogen.o: hydrogen.cpp eigenval.h outputarr.h initmat.h
eigenval.o: eigenval.cpp eigenval.h
outputarr.o: outputarr.cpp outputarr.h
initmat.o: initmat.cpp initmat.h
hydrogen: hydrogen.o eigenval.o outputarr.o initmat.o
	$(CXX) $(CXXFLAGS) -o $@ $^
clean: ; $(RM) hydrogen.o eigenval.o outputarr.o initmat.o
```

```cpp
// outputarr.h
#ifndef OUTPUTARRH
#define OUTPUTARRH
#include <string>
#include <rarray>
void writeBinary(const std::string& name,
                 const rvector<double>& a);
void writeText(const std::string& name,
               const rvector<double>& a);
#endif
```

```cpp
// outputarr.cpp
#include "outputarr.h"
#include <fstream>
void writeBinary(const std::string& name,
                 const rvector<double>& a) {
    std::ofstream g(name, std::ios::binary);
    g.write((char*)(&a[0]), a.size()*sizeof(a[0]));
    g.close();
}
void writeText(const std::string& name,
               const rvector<double>& a) {
    std::ofstream f(name);
    for (int i = 0; i < a.size(); i++)
        f << a[i] << std::endl;
    f.close();
}
```

Example: Integrated test for hydrogen

Save the original (monolythic) code, and run it, moving output to other file:

```
$ g++ -std=c++17 -O2 -g -o hydrogen_monolythic hydrogen_monolythic.cpp
$ ./hydrogen_monolythic > hydrogen_monolythic.out
$ mv data.txt data_monolythic.txt
$ mv data.bin data_monolythic.bin
```

Run the modular code:

```
$ make hydrogen
$ ./hydrogen > hydrogen.out
```

Compare the outputs:

```
$ diff hydrogen.out hydrogen_monolythic.out
$ diff data.txt data_monolythic.txt
$ cmp data.bin data_monolythic.bin
```

This is a very good idea when modularizing code, because you cannot do unit tests yet.

Warning: the byte-for-byte comparison can break for floating point numbers.

Example: Unit test for outputarr module

```
#include "outputarr.h"
#include <iostream>
#include <fstream>
int main() {
   std::cout << "A UNIT TEST FOR FUNCTION 'writeText'\n";
   // create file:
   rvector<double> a(3);
   a = 1, 2, 3;
   writeText("testoutputarr.txt", a);
   // read back
   std::ifstream in("testoutputarr.txt");
   std::string s[3];
   in >> s[0] >> s[1] >> s[2];
   // check
   if (s[0]!="1" or s[1]!="2" or s[2]!="3") {
       std::cout << "TEST FAILED\n";
       return 1;
   }  else {
       std::cout << "TEST PASSED\n";
       return 0;
   }
```

Add to makefile:

```
...
test: outputarr_test
   ./outputarr_test

output_arr_test: outputarr_test.o outputarr.o
   $(CXX) -g -O2 -o $@ $^

output_arr_test.o: outputarr_test.cpp outputarr.h
```

To run:

```
$ make test
g++ -std=c++17 _O2 -c -o outputarr_test.o outputarr_test.cpp
g++ -o outputarr_test outputarr_test.o outputarr.o
./outputarr_test
A UNIT TEST FOR FUNCTION 'writeText'
TEST PASSED
$ echo $?
0
```

Guidelines for testing

- Each module should have a separate test suite
  (so outputarr_test.cc should also have a test for `writeBinary`).
- If the code is properly modular, those module test should not need any of the other .cpp files.
- Testing will give confidence in your module, and will tell you which modules have stopped working properly.
- Once your tests are okay, you now have a piece of code that you could easily use in other applications as well, and which you can comfortably share.

Testing frameworks

- There's a lot of extra coding here just to run the tests.
- The tests need to be maintained as well.
- Especially when your project contains a lot of tests, you may want to use a unit testing framework.

Examples:

- Boost.Test (from the Boost library suite)
- Google C++ Testing Framework (a.k.a googletest)
- Catch2
- ...

These are typically combinations of macros, a driver main function that can select which tests to run, etc.

- For the assignment, if you're going to use a framework, use Boost.Test.

Example of Boost.Test

```cpp
// output_bt.cpp
#include "outputarr.h"
#include <iostream>
#include <fstream>

#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE output_bt
#include <boost/test/unit_test.hpp>
BOOST_AUTO_TEST_CASE(writeText_test)
{
    // create file:
    rvector<double> a(3);
    a = 1,2,3;
    writeText("testoutputarr.txt", a);
    // read back:
    std::ifstream in("testoutputarr.txt");
    std::string s[3];
    in >> s[0] >> s[1] >> s[2];
    // check
    BOOST_CHECK(s[0]=="1" && s[1]=="2" && s[2]=="3");
}
```

```
$ module load gcc/9 boost

$ g++ -std=c++17 -c -g -O2 output_bt.cpp

$ g++ -g -O2 -o output_bt output_bt.o outputarr.o \
   -lboost_unit_test_framework

$ ./output_bt --log_level=all
```

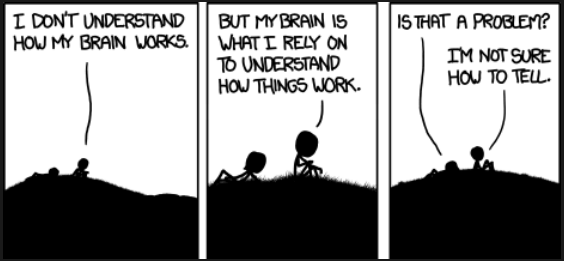
```
Running 1 test case...
Entering test module "output_bt"
output_bt.cpp(10): Entering test case "writeText_test"
output_bt.cpp(21): info:
  check s[0]=="1" && s[1]=="2" && s[2]=="3" has passed
output_bt.cpp(10):
  Leaving test case "writeText_test"; testing time: 493us
Leaving test module "output_bt"; testing time: 521us

*** No errors detected
```

3

Debugging

What if your program or test isn't running correctly...

- Nonsense. All programs execute "correctly".
- We just told it to do the wrong thing.
- Debugging is the *art* of reconciling your mental model of what the code is doing with what you actually told it to do.



https://imgs.xkcd.com/comics/debugger.png

Debugger: program to help detect errors in other programs.

Tips to avoid debugging

- Write better code.
  - simple, clear, straightfoward code.
  - modularity (avoid global variables and 10,000 line functions).
  - avoid "cute tricks", (**no** obfuscated C code winners – IOCCC).
- Don't write code, use existing libraries.
- Write (simple) tests for each module.
- Use version control and small commits.
- Switch on the `-Wall` flag, inspect all warnings, fix them or understand them all.
- Use defensive programming:

  Check arguments, use assert (which can be switched of with `-DNDEBUG` compilation flag) E.g.:

```
#include <cassert>
#include <cmath>
double mysqrt(double x) {
    assert(x>=0);
    return sqrt(x);
}
```

Debugging workflows

- As soon as you are convinced there is a real problem, create the simplest situation in which it repeatedly occurs.
- This is science: model, hypothesis, experiment, conclusion.
- Try a smaller problem size, turning off different physical effects with options, etc, until you have a simple, fast, repeatable example.
- Try to narrow it down to a particular module/function/class.
- Integrated calculation: Write out intermediate results, inspect them.

Despite that, still errors?

Some common issues:

Arithmetic	Corner cases (`sqrt(-0.0)`), infinities
Memory access	Index out of range, uninitialized pointers
Logic	Infinite loop, corner cases
Misuse	Wrong input, ignored error, no initialization
Syntax	Wrong operators/arguments
Resource starvation	Memory leak, quota overflow
Parallel	Race conditions, deadlock

To figure out what is going wrong, and where in the code, we can

- Put strategic print statements in the code.
- Use a debugger.

What's wrong with using print statements?

Strategy

- Constant cycle:
  - strategically add print statements
  - compile
  - run
  - analyze output
  - repeat
- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems with this approach

- Time consuming
- Error prone
- It changes memory, timing ...

There's a better way!

Debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Use a graphical debugger or not?

- Local work station: graphical is convenient
- Remotely (SciNet): can be slow or hard to set up.
- In any case, graphical and text-based debuggers use the same concepts.

Debuggers

Preparing the executable

- Add required compilation flags, `-g` (sometimes `-g -gstabs`) (both in compiling and linking!)
- Optional: switch off optimization `-O0`

Command-line based symbolic debuggers: gdb

- Free, GNU license, symbolic debugger.
- Available on many systems.
- Been around for a while, but still developed and up-to-date
- Text based, but has a `-tui` option.

```
$ module load gdb
$ gdb -tui app
...
(gdb)_
```

GDB command summary

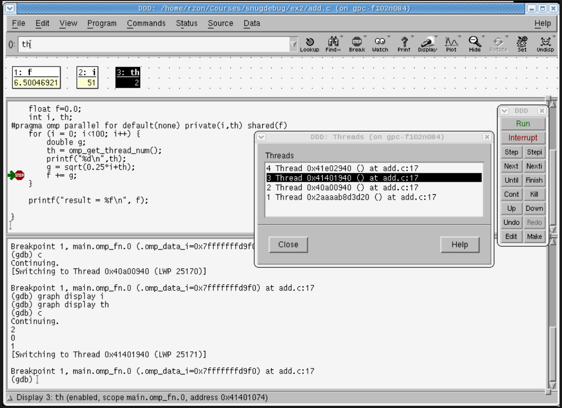
`help`	`h`	print description of command
`run`	`r`	run from the start (+args)
`backtrace/where`	`ba`	function call stack
`break`	`b`	set breakpoint
`delete`	`d`	delete breakpoint
`continue`	`c`	continue
`list`	`l`	print part of the code
`step`	`s`	step into function
`next`	`n`	continue until next line
`print`	`p`	print variable
`display`	`disp`	print variable at every prompt
`finish`	`fin`	continue until function end
`set variable`	`set var`	change variable
`down`	`do`	go to called function
`until`	`unt`	continue until line/function
`up`	`up`	go to caller
`watch`	`wa`	stop if variable changes
`quit`	`q`	quit gdb

Demonstration:

Let's see how to use gdb.

Graphical debuggers

DDD: free, bit old, can do serial and threaded debugging.



DDT: commercial, on SciNet, good for parallel debugging (including mpi and cuda)