Testing and Debugging (PHY1610 Lecture 8)

Ramses van Zon

SciNet HPC Consortium

January 31, 2017

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 a 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

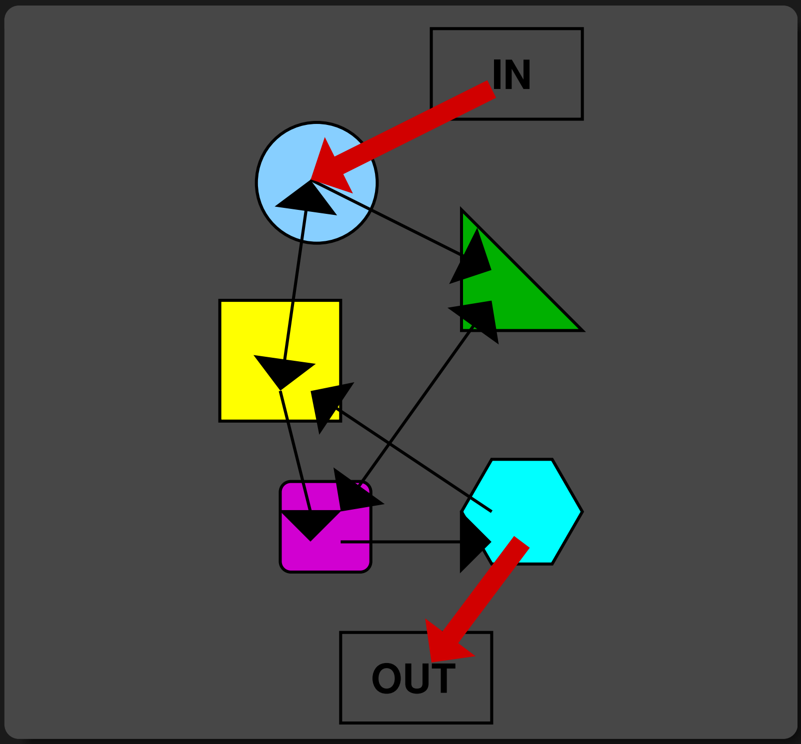
1. **How do we ensure a module works correctly?**
   $\Rightarrow$ Unit testing

2. **What if we find that it doesn't?**
   $\Rightarrow$ Debugging

Unit testing

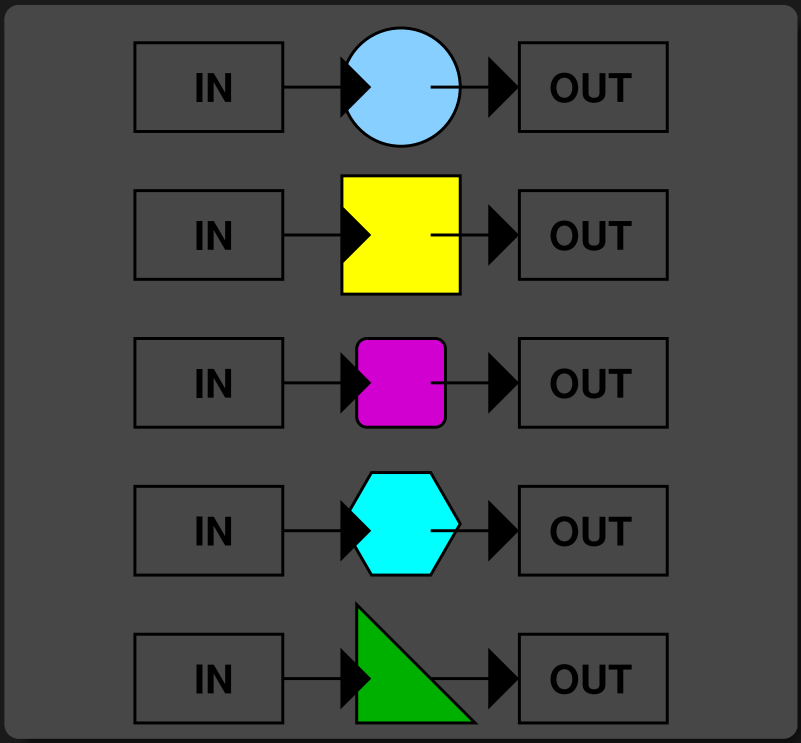
Integrated testing

- 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

- The 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 3 (unmodular)

```
// hydrogen_monolyth.cc
#include <rarray>
#include <iostream>
#include <fstream>
#include <cmath>
const int n = 100;
rmatrix<double> m(n,n);
rvector<double> a(n);
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[i];
  a = q.copy();
}
double en() {
  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[i];
  double e=0.0, z=0.0;
  for (int i=0;i<n;i++) {
      e += a[i] * q[i];
      z += a[i] * a[i];
  }
```

```
  return e/z;
}
int main() {
  a.fill(1);
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      m[i][j] = 1.0/(1.0+fabs(i*i-j*j));
  double b = 0;
  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<10;p++)
    pw();
  for (int i=0; i<n; i++)
    m[i][i] += b;
  std::cout<<"Ground state energy is "<<en()<<std::endl;
  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.data()),a.size()*sizeof(a[0]));
  return 0;
}
```

Example from lecture 3 (modular)

```
// hydrogen.cc
#include <rarray>
#include <iostream>
#include "outputarr.h"
#include "initmat.h"
#include "eigenvals.h"
int main() {
  const int n = 100;
  rmatrix<double> m(n,n);
  rvector<double> a(n);
  initmat(m);
  double en = ground_state(m,a);
  std::cout<<"Ground state energy is "<<en<<std::endl;
  toAsc("data.txt", a);
  toBin("data.bin", a);
  return 0;
}
```

```
# Makefile
CXXFLAGS=-g -std=c++11
all: hydrogen
hydrogen.o: hydrogen.cc eigenvals.h outputarr.h initmat.h
outputarr.o: outputarr.cc outputarr.h
initmat.o: initmat.cc initmat.h
eigenvals.o: eigenvals.cc eigenvals.h
hydrogen: hydrogen.o initmat.o eigenvals.o outputarr.o
  $(CXX) -g -o hydrogen hydrogen.o initmat.o eigenvals.o outputarr.o
```

```
// outputarr.h
#ifndef OUTPUTARR_H
#define OUTPUTARR_H
#include <string>
#include <rarray>
void toBin(std::string& s, rarray<double,1>& x);
void toAsc(std::string& s, rarray<double,1>& x);
#endif
```

```
// outputarr.cc
#include "outputarr.h"
#include <fstream>

void toBin(std::string& s, rarray<double,1>& x) {
  std::ofstream g(s,std::ios::binary);
  g.write((char*)(x.data()),x.size()*sizeof(x[0]));
  g.close();
}

void toAsc(std::string& s, rarray<double,1>& x) {
  std::ofstream f(s);
  for (int i=0; i<x.size(); i++)
    f << x[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++11 -o hydrogen_monolythic hydrogen_monolythic.cc
$ ./hydrogen_monolythic > hydrogen_monolythic.out
$ mv data.bin data_monolythic.bin
$ mv data.txt data_monolythic.txt
```

Run the modular code:

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

Compare the output:

```
$ 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

```
// outputarr_test.cc
#include "outputarr.h"
#include <iostream>
#include <fstream>
int main() {
  std::cout << "UNIT TEST FOR FUNCTION 'toAsc'\\n";
  // create file:
  rvector<double> a(3);
  a = 1,2,3;
  toAsc("testoutputarr.txt", a);
  // read back:
  std::ifstream in("testoutputarr.txt");
  std::string s1,s2,s3;
  in >> s1 >> s2 >> s3;
  // check:
  if (s1!="1" or s2!="2" or s3!="3") {
    std::cout << "TEST FAILED\n";
    return 1;
  } else {
    std::cout << "TEST PASSED\n";
    return 0;
  }
}
```

Add to makefile:

```
#Makefile
...
test: outputarr_test
	./outputarr_test
outputarr_test: outputarr_test.o outputarr.o
	$(CXX) -g -o outputarr_test outputarr_test.o outputarr.o
outputarr_test.o: outputarr_test.cc outputarr.h
```

To run:

```
$ make test
g++ -g -std=c++11 -c -o outputarr_test.o outputarr_test.cc
g++ -o outputarr_test outputarr_test.o outputarr.o
./outputarr_test
UNIT TEST FOR FUNCTION 'toAsc'
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 toBin).
- If the code is properly modular, those module test should not need any of the other .cc 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)
  - ...

  These are typically combinations or 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

```
// output_bt.cc
#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(toAsc_test)
{
  // create file:
  rvector<double> a(3);
  a = 1,2,3;
  toAsc("testoutputarr.txt", a);
  // read back:
  std::ifstream in("testoutputarr.txt");
  std::string x,y,z;
  in >> x >> y >> z;
  // check:
  BOOST_CHECK(x=="1"&&y=="2"&&z=="3");
}
```

```
$ g++ -std=c++11 -g -c output_bt.cc
$ g++ -g -o output_bt output_bt.o outputarr.o\
      -lboost_unit_test_framework
$ ./output_bt --log-level all
```

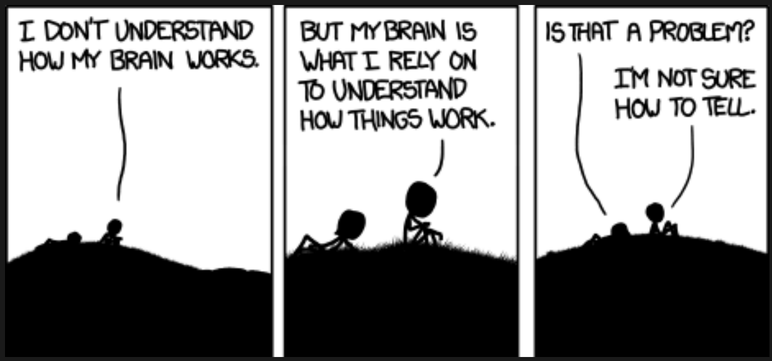
```
Running 1 test case...
Entering test suite "output_bt"
Entering test case "toAsc_test"
output_bt.cc(19): info: check x=="1"&&y=="2"&&z=="3" passed
Leaving test case "toAsc_test"; testing time: 1036mks
Leaving test suite "output_bt"

*** No errors detected
```

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.



http://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.
- 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`).

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

SciNet Physics UNIVERSITY OF TORONTO

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

1. Put strategic print statements in the code.
2. 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
- 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
- In any case, graphical and text-based debuggers use the same concepts.

Debuggers

Preparing the executable

- Add required compilination flags:

```
$ g++ -g -gstabs code.cc -o app
```

- 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.

```
$ gdb app
...
(gdb)_
```

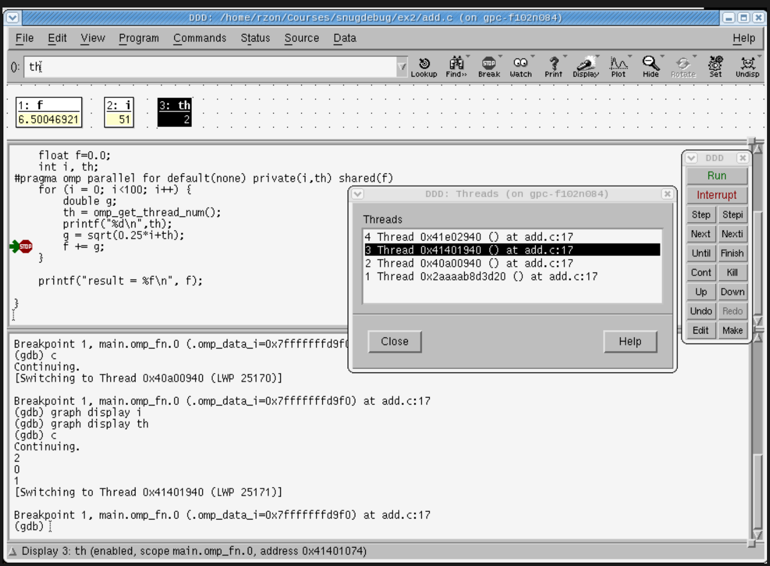
GDB command summary

Demonstration:

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

Graphical debuggers

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



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