Numerical Tools for Physical Scientists: Modelling, Validation, & Verification

Erik Spence

SciNet HPC Consortium

4 February 2014

This course

We covered the basics of C++ and good programming practices in the last course. In this course we are going to focus on the specifics of numerical computing for physical scientists. If there is a specific numerical technique that you'd like us to discuss, tell us and we'll see if we can work it in.

This course will cover:

- Modelling, Validation, Verification.
- Random numbers and Monte Carlo.
- Optimization, root finding.
- ODEs and molecular dynamics.
- Linear Algebra.
- Fast Fourier Transform.

Today's class

Today we will cover:

- What is computational science?
- Verification and validation?
- Universal errors.

Computational science

Computational science is a relatively new approach to science.

- It is often called the "third leg" of science, the other two being experiment and theory.
- It is different from theoretical or experimental science, but requires the skills of both:
  - it requires the note-taking, methodical approach of experimentalists;
  - it requires the mathematical skills of theorists;
  - it requires its own expertise in programming, algorithms, numerical stability, computer science, etc.
- Computational science is usually done very badly. Why? Its interdisciplinary nature has resulted in few that have the expertise to do it properly.

Computational science means modelling

Science is the *empirical* study of the natural world.

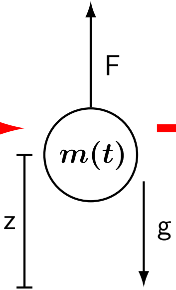
Computational science is an exercise in modelling the natural world.

- Note the difference: we're not talking about crunching data, for which computing is also used, and to which many of the techniques you will learn can be applied.
- We are talking about building models, based on theory or law, and using them to make predictions about the natural world, or better understand existing observations.
- Computational science is not reality! Computation science is used to create *models*. Experimental science is reality.

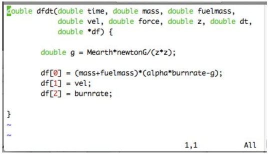
Basic Framework

Data

Laws/Theory



Numerical Computation



```
double dfdt(double time, double mass, double fuelmass,
            double vel, double force, double z, double dt,
            double *df) {

        double g = Mearth*newtonG/(z*z);

        df[0] = (mass+fuelmass)*(alpha*burnrate-g);
        df[1] = vel;
        df[2] = burnrate;

}
~
~
                                          1,1           All
```

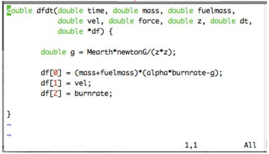
$$m(t)\ddot{z} = F - m(t)g$$
$$F \propto \dot{m}$$

What can go wrong?

- Problems going from data to laws/theory: beyond the scope of this class.
- We can have errors translating from a mathematical law/theory to a computational model.
- Various types of computation errors can be introduced:
    - Discretization error.
    - Truncation error.
    - Roundoff error.
    - Numerical instabilities.
- Or just plain old bugs can be introduced.
- Process of testing this: verification and validation.

$$m(t)\ddot{z} = F - m(t)g$$

$$F \propto \dot{m}$$



```
double dfdt(double time, double mass, double fuelmass,
            double vel, double force, double z, double dt,
            double *df) {

          double g = Mearth*newtonG/(z*z);

          df[0] = (mass+fuelmass)*(alpha*burnrate-g);
          df[1] = vel;
          df[2] = burnrate;

}
~
~
                                                1,1           All
```

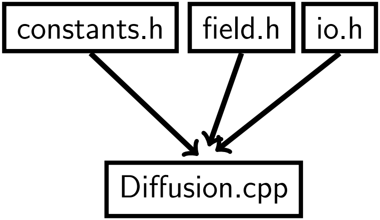
Verification: test!

Verification is the act of confirming that we are solving the intended equations correctly in the regime of interest. This requires testing:

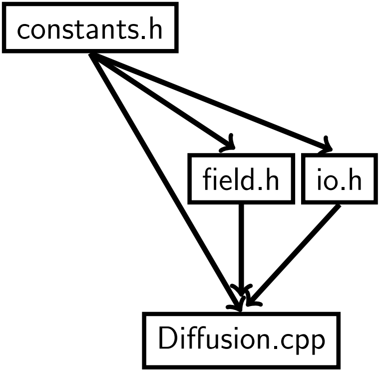
- Use modular programming.
- Develop code whose sole purpose is to test the module that you are developing. I usually put it in a separate 'testing' directory.
- Add the testing code to the repository.
- Compile the testing code with an optional rule in your Makefile.
- Develop many and manifold tests. Any unique and relevant test you can think of, put it in there.
- As a general rule, unless you've tested the code recently, assume it doesn't work.
- If someone gives you code that doesn't come with testing functions, assume it doesn't work (and assume he doesn't know how to write good code).

Modularity makes testing much easier

Good



Not Good



Verification: analytics and benchmarks

So we're writing our testing code. What kind of tests can we perform?

- Comparison to analytic solutions:
    - Solutions tend to be for simple situations - not hard tests of the computation.
    - Necessary. If your code doesn't solve these correctly you've got serious problems.
    - Analytic solutions exist for certain classes of resolution tests, since you have control over the size of your error.
- Benchmarking: comparing the results of your code to other codes which solve the same problem, in the same parameter regime.
    - Does not demonstrate that either solution is correct.
    - Can show that at last one code or version has a problem, or that something has caused changes.
    - Is more powerful if different algorithm types are used.
    - Save the results of benchmarks in your testing directory.

Verification: convergence

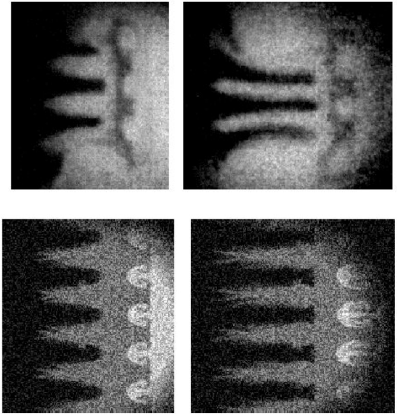
Convergence testing is used to answer the question: is my resolution high enough?

- Convergence testing is performed by increasing the resolution of the code and confirming that the result does not change significantly.
- For codes that include expansions of quantities (such as spectral or pseudo-spectral codes), convergence testing includes increasing the number of terms of expansion to see if the result changes significantly.
- What does "change significantly" mean? That depends on what you're studying. If you're not sure, ask someone who knows. Different fields have different criteria.
- Do not fall into the trap of not doing convergence studies!
- Again, the existence of convergence does not mean that the solution is correct, but lack of convergence indicates a problem.

Validation: testing against reality

The only way to know that enough natural law has been incorporated into the model is to test it against the real world, which means data.

- The only way to do validation of experimental simulations is to compare to the experiment that you're simulating.
- It must be in a regime you are realistically interested in, but still experimentally accessible.
- It requires collaboration with experimenters.
- It demonstrates that there is a regime in which your code successfully reproduces reality.

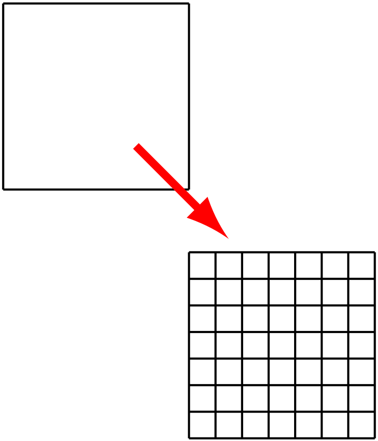


arXiv:astro-ph/0206251

Discretization error

What is it? Where does it come from?

- In the real world space and time are continuous. But simulations and calculations are not.
- Variables must be converted from continuous to discrete.
- Space is sliced up into grids. Time is changed to steps.
- The density of the grids and steps goes up with increasing resolution.



Discretization error, continued

Discretization error is the error introduced to a calculation by the act of discretizing the variables. What's the problem?

- One must be sure the grid density (resolution) is high enough that discretization errors are at an acceptable level.
- One must be sure that the resolution is high enough that all features of the physical system are being captured by the computation.
- What resolution is high enough? This depends on what is being discretized (time versus space), the type of calculation, and other factors.
- There are relationships between the discretization of the various variables that need to be respected, to keep discretization errors under control (and to prevent numerical instabilities).

Truncation error

Truncation error occurs when an expansion in your calculation is truncated. Meaning, instead of using this:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

we use this:

$$e^x \simeq 1 + x + \frac{x^2}{2}.$$

Obviously truncation is necessary. How do we determine where to truncate? How many terms should we keep?

Where to truncate?

Choosing where to truncate is sometimes more art than science. The question you need to answer: is what I am throwing away important to the calculation?

Sometimes the answer is obvious. In the case of e^x, we can sensibly truncate when we reach machine precision, meaning choose n such that

$$\left|\frac{x^n}{n!}\right| < \epsilon$$

where ϵ is machine precision.

Other cases are not so obvious. Here's how I approach the problem: determine some metric for what you are expanding which captures its importance (size, magnitude, energy, ...) and then compare the largest term you are throwing away to the largest non-trivial term. I like to have at least one order of magnitude size difference between them, preferably two orders.

Roundoff errors

Roundoff error occurs when you're not being careful with which combinations of types of numbers you are operating on:

$$(a+b)+c \neq a+(b+c)$$

```cpp
#include <iostream>          // RoundOff.cpp
int main() {
  double a = 1.0, b = 1.0, c = 1e-16;

  std::cout << (a - b) + c << std::endl;
  std::cout << a - (b + c) << std::endl;
  return 0;
}
```

Roundoff errors

Roundoff error occurs when you're not being careful with which combinations of types of numbers you are operating on:

$$(a+b)+c \neq a+(b+c)$$

```
#include <iostream>          // RoundOff.cpp
int main() {
  double a = 1.0, b = 1.0, c = 1e-16;

  std::cout << (a - b) + c << std::endl;
  std::cout << a - (b + c) << std::endl;
  return 0;
}
```

```
ejspence@mycomp ~> g++ RoundOff.cpp -o RoundOff
ejspence@mycomp ~> ./RoundOff
1e-16
0
```

Roundoff errors, continued

Roundoff errors can occur anytime you start operating near machine precision.

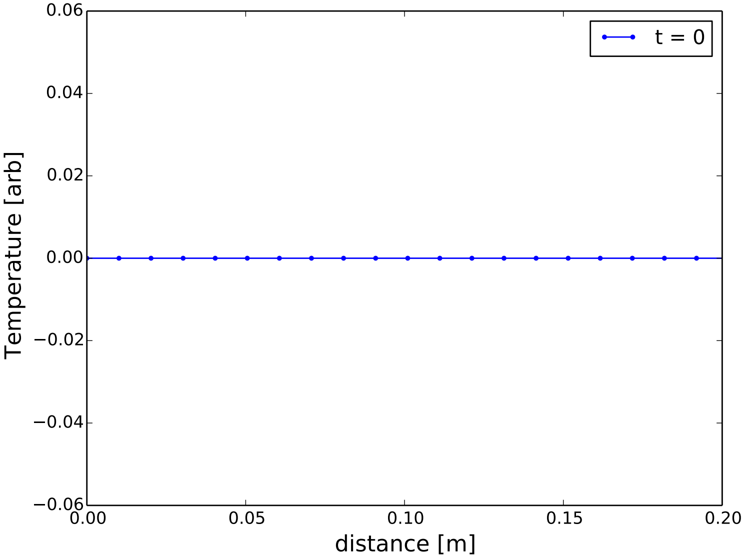
- 'Machine precision' (or 'machine epsilon') is the upper bound on the relative error due to rounding. This is generally 1e-8 for single precision (float) and 1e-16 for double.
- Roundoff errors are most common when subtracting or dividing two non-integer numbers that are about the same size, thus forcing the computer to do arithmetic near machine epsilon.
- Do your best to modify your algorithms to avoid such calculations.

Numerical instabilities

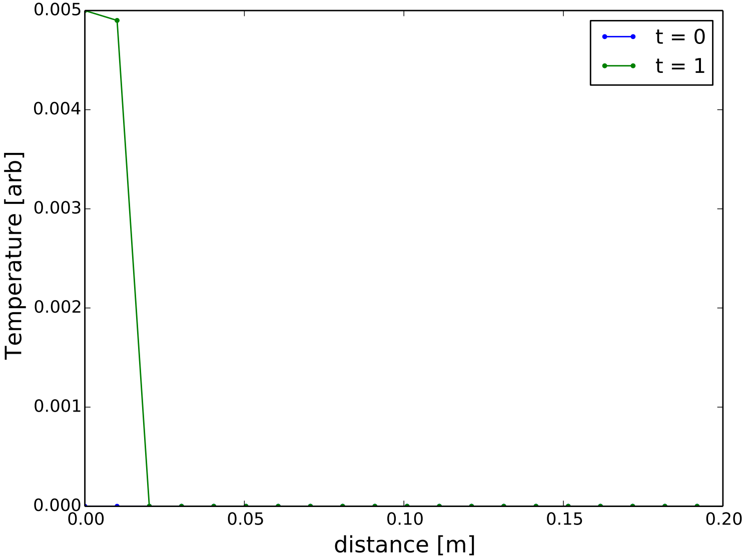
Huh? What are numerical instabilities?

- Numerical instabilities happen in a variety of situations. You'll know you have an instability when things 'blow up'.
- But how can that happen? Don't my equations represent reality? Reality doesn't 'blow up'.
- The problem is that reality is continuous, and we've discretized the problem.
- How do we avoid instabilities? Usually high-enough resolution, in space or time or both, will prevent instabilities.
- Certain classes of algorithms and techniques are more prone to instabilities than others. Be aware of the weaknesses in the algorithm you're using.
- Be aware of what they look like.

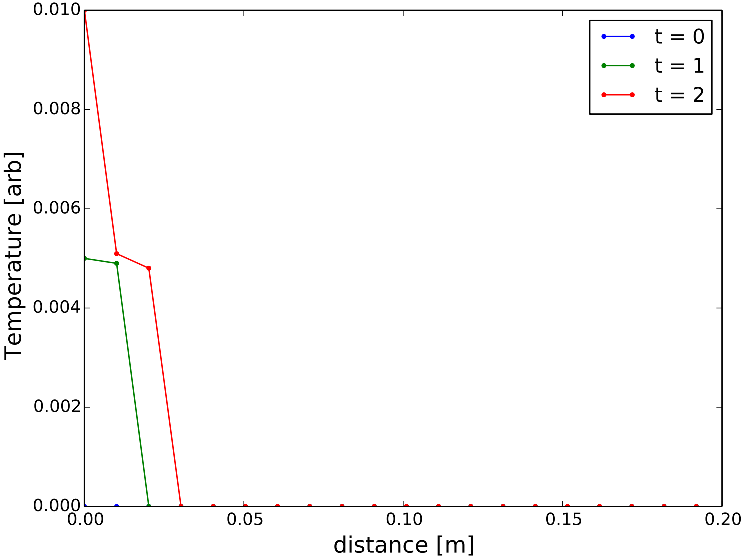
Numerical instability example



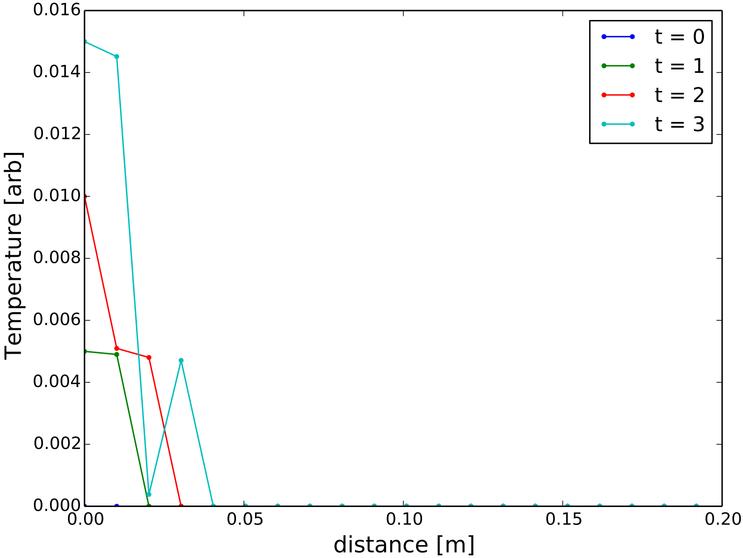
Numerical instability example



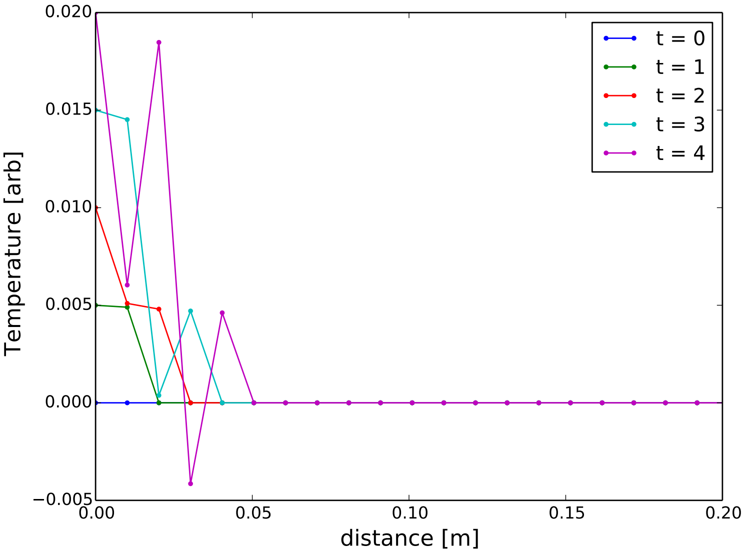
Numerical instability example



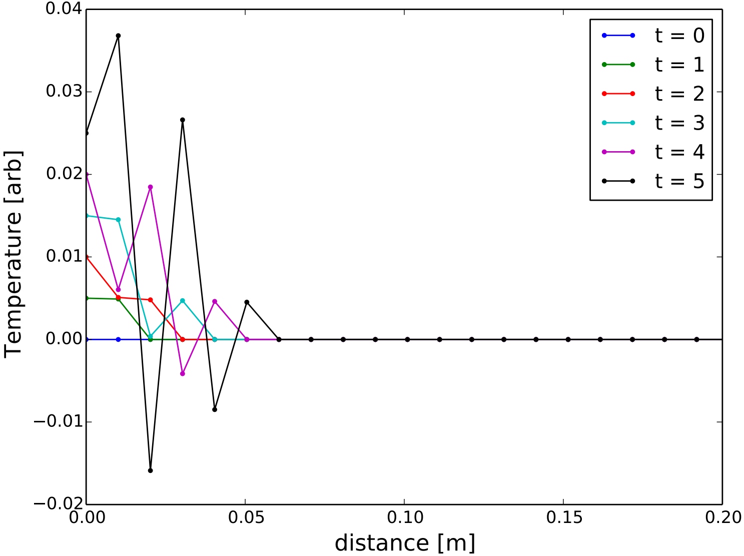
Numerical instability example



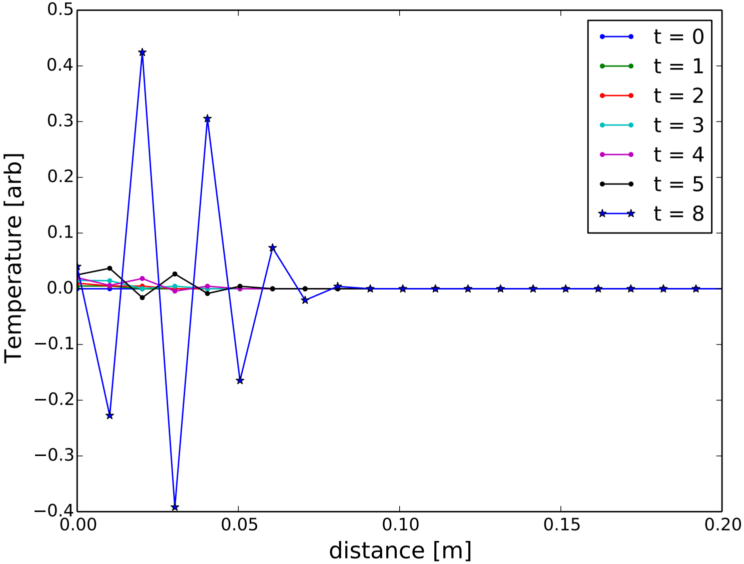
Numerical instability example



Numerical instability example



Numerical instability example



Numerical instability example