PART I - Numerical Software

Raymond J. Spiteri

Lecture Notes for CMPT 898:
Numerical Software

University of Saskatchewan

January 7, 2013

Objectives

- What this course is about.
- What this course isn't about.
- Power tools of the trade.

What is numerical software?

Numerical software refers to software that does computations with real numbers.

Such computations are commonly used by scientists and engineers, especially when modelling continuous quantities such as velocity fields, charge densities, and material stresses.

Because computers are finite machines, we must approximate the desired continuous quantities with discrete ones.

This leads us to the study of floating-point numbers as they mean to approximate real numbers.

In particular, two concepts play a critical role in numerical software: round-off error, the difference between a real number and its floating-point representation, and numerical stability, how round-off errors propagate through an algorithm.

What is numerical software?

Numerical software is normally intended to investigate the solution large-scale problems.

The definition of “large scale” is a moving target; it refers to something in the ball park of what the current supercomputers would consider realistically feasible.

As of today (2013), we might think of large scale as trying to solve for 10 million unknowns.

By definition, when trying to model continuous quantities, there is always an uncountable infinity of values of which to keep track.

Clearly, some form of discretization is required: a way to approximate continuous (infinite) quantities and operators with discrete (finite) ones.

It is not hard to imagine that the finer the discretization, the better the approximation.

What is numerical software?

Let's consider the following example, not only to see how large a problem might be but also how large it can get in a short amount of time.

The human heart has approximately 10^{10} muscle cells.

Each cell may need to have $\mathcal{O}(10)$ state variables.

This leads to a total of $\mathcal{O}(10^{11})$ unknowns.

Suppose we wish to have a simulation of a few minutes, i.e., $\mathcal{O}(10^2)$ heartbeats (or s).

To achieve sufficient resolution in time, this may require time steps of size $\mathcal{O}(10^{-5})$ s or $\mathcal{O}(10^7)$ steps.

In summary, the goal of a single simulation is to propagate $\mathcal{O}(10^{11})$ unknowns through $\mathcal{O}(10^7)$ steps.

(And to perform an entire study, $\mathcal{O}(10^3)$ simulations like this must be performed.)

What is numerical software?

Because of the large numbers of quantities involved, efficiency in time and memory is a crucial consideration.

An often precarious balance must be struck between

- the fidelity (complexity) of the model,
- the resolution of the discretization, and
- the efficiency of the algorithm

in order to achieve results with the resources (time, computing, human) available.

Power tools of the trade

Nowadays, the most prevalent power tool of the trade in numerical software is parallel computing.

There are only two (potentially) legitimate reasons to write a parallel program to solve a given problem:

- The memory required is more than is available on a single computer.
- The program takes “too long” to run.

But beware:

Parallel programming is easy as long as you don’t care about performance.

Power tools of the trade

The most popular (and powerful) programming languages for numerical computing are

1. Fortran
2. C / C++
3. Python
4. MATLAB
5. Java

Power tools of the trade

Fortran was developed to be at just high enough a level to make programming humane.

The executables were lean and close to machine language, making them highly efficient.

C is an extremely popular, general-purpose language that compiles efficiently to machine instructions. C is a *procedural* language.

As advertized, C^{++} is everything C is and more. It is generally used for its *object-oriented* capabilities.

Python is a general-purpose programming language that is gaining in popularity. It is interpreted and often used as a scripting language. It was designed to be even an higher-level language than Fortran or C/C^{++}, with syntax that is meant to be *clear and expressive*.

Perhaps Python's most famous claim to fame (in the context of numerical software) is its easy interoperability with Fortran and C.

Strictly speaking, MATLAB is a problem-solving environment, but it is also a programming language.

MATLAB programs are generally interpreted and thus perform much more slowly than compiled programs. However, MATLAB code can be compiled into C++ code to enhance performance.

When Java first appeared, features such as automatic garbage collection made Java significantly slower than C/C++ or Fortran.

Since the advent of the virtual machine, the difference is not necessarily that large.

Power tools of the trade

Other programming languages that are amenable to numerical software:

- Julia
- Fortress
- R

Of these, R seems to be surging in popularity.

A consistent theme seems to be to enhance programmer quality of life (make code intuitive, automatic parallelization, catch obvious mistakes like index out of bounds, deallocate memory, etc.) while at the same time have efficient execution.

Problem-solving environments

Problem-solving environments (PSEs) are specialized software environments that are intended to make the general computational scientist's life easier by providing easy access to high-quality numerical libraries through a friendly interface.

The idea is that the scientist can focus on the science without getting bogged down by programming details.

Of course there is also the potential spin-off benefit that better methods will be used through the use of the PSE libraries!

The classical examples of PSEs for general mathematically formulated computational problems are MATLAB, Maple, and Mathematica.

However, other packages also exist in various other more specialized contexts, including Comsol Multiphysics, FEniCS, LabView, paraview, pythNon, odeToJava, Berkeley Madonna, and VenSim.

Problem-solving environments

Arguably PSEs have become the *modus operandi* of most computational scientists and engineers.

The main advantages are:

- easy to prototype, experiment, visualize results
- advanced solution methods
- automatic method selection

The main disadvantages are:

- it is really hard to be all things to all people
- sub-optimal performance

In numerical software, we generally trade performance for convenience, but only to a point.

Software and science

The typical approach to writing numerical software mirrors the process of scientific discovery:

- Start with a(n interesting) problem.
- Find a reasonable algorithm.
- Write some code; check it; see what happens.
- Make the problem harder; make the code better.

Simulation is now considered on par with theory and experiment as a pillar of science.

This means the results produced by software needs to be taken seriously in terms of its accuracy, sophistication, and reproducibility.

Software and science

Because the first thing you try is unlikely to be an earth-shattering discovery, numerical software must be flexible enough to still be useful even after the question is changed (somewhat).

The challenge of such rapidly changing requirements highlights the importance of having a reasonably flexible software design.

An important principle in numerical software is to maintain a balance of good practice with correctness and feasibility.

Correctness and feasibility are obviously non-negotiable; without them, the software is useless.

Good practice, however, allows the software to be useful for a longer period of time (i.e., mitigating the necessity to start from scratch on new software) and to more people (e.g., your collaborators).

What this course is not about

Although numerical software . . .

- often greatly benefits from the principles of software engineering (SWE), this is not a course about SWE.
- often uses or produces copious amounts of data, this is not a course about data processing.
- often requires results to be visualized to make sense of them, this is not a course about visualization.
- is usually based on sound numerics, this is not a course about numerical analysis.
- often runs on high-performance computers, this is not a course about parallel computing.

In general, our approach will be to learn only enough about a given topic to be dangerous; i.e., to derive 80% of the benefit of the knowledge while expending only 20% of the effort required to become an expert.

Highlights of the numerical software process

It is useful to divide the process of producing numerical software into four steps, which are not necessarily followed in a linear progression:

1. determine problem specifications

2. design and analyze algorithm

3. implement code

4. verify and validate the results

It should also be noted that none of these steps are usually trivial.

Highlights of the numerical software process

Ideally the specification is a precise and complete statement of the intended relationship between the problem statement (input) and the solution (output).

For example, given a quadratic equation, find its roots.

Even this simple problem is fraught with subtleties!

- What if there are no (real) roots?
- Are the coefficients real?
- Are there errors in the coefficients?
- What defines a solution?

Highlights of the numerical software process

The algorithm is the sequence of steps to be taken to go from input to output.

The steps may also consist of sub-steps that are problems in themselves.

It is generally important to never even try to re-invent the wheel; it makes sense to use high-quality libraries whenever possible.

In numerical software, we generally assume that input data are unexceptional.

It will always be possible to design artificial problems that will break any algorithm.

But if such problems do not occur in nature, writing software to handle them is not that useful.

This does not mean input checking is not important!

Highlights of the numerical software process

The implementation of the algorithm is ultimately meant to working code.

Working code is not simply code that compiles.

It is impossible to determine whether numerical software works without performing some verification and validation.

In some sense, the whole point of writing numerical software is to address questions whose answers we do not know a priori.

As is the nature of science, it is not generally possible to know whether a piece of numerical software is correct.

It is only possible to know that it is not incorrect, or at least not inconsistent with other evidence.

Highlights of the numerical software process

The verification and validation of the results of numerical software are of crucial importance yet they are often not emphasized.

Computers always do what they are told; the challenge of programming is to ensure they are doing what you *intend* them to do.

Verification refers to determining whether the software is performing as intended in a literal sense.

Even if software is executing as envisioned at a low level, its results may not match reality.

Validation refers to whether the software produces results that are consistent with data obtained by independent means, e.g., observation.

Arguably, validation is much less about programming than modelling.

Priorities

Without question, the priorities when writing scientific software are (in order):

1. correctness of results

2. numerical stability

3. accuracy of discretizations; error estimation

4. flexibility of software

5. efficiency (execution speed and memory)

Obviously, if the software does not produce correct results, then nothing else matters.

If the algorithm is not numerically stable, the results cannot be trusted (nor are they likely to be correct).

Priorities

If the discretizations are not accurate, the results cannot be trusted (nor are they likely to be correct).

It is generally not meaningful to give a result without some quantification of its uncertainty or error.

If the software is not at least somewhat flexible, its useful lifetime is likely to be severely curtailed as problems, programming languages, and computer hardware evolve.

It is also likely no one else will want to bother to use your software.[1]

Efficiency is listed last, but it is not necessarily least.

It is true that if satisfactory results can be obtained, efficiency is often treated as superfluous.

However if the computation is simply not feasible with the hardware / algorithm configuration at your disposal, then efficiency also becomes indispensable.

[1]It is not always clear whether this is good or bad!

Efficiency

As mentioned, the approach to efficiency in numerical software is taken with feasibility in mind.

Code can be executed many times in large-scale problems, so any little inefficiency can be magnified to the point of making the computation infeasible.

As computational scientists, we tend to balance low-level code optimization against readability, flexibility, and portability.

This is in contrast to some fields, such as graphics, games, or real-time systems, where parameters such as hardware are fixed.

In such cases, code optimization is necessary; often every last bit of performance must be squeezed out of a chip for all the desired effects to be achieved.

Some knowledge of computer architecture is generally necessary in order to design effective numerical software, but usually only general principles.

Essence of software optimization over hardware

The main bottleneck in computing is the disparity between the time it takes a process to fetch data compared to the time it takes to manipulate data.

The following is the list of where data are stored (in increasing size, cost, and access time):

- registers: small fast storage buffers within the CPU
- cache (and various levels thereof): small fast memory located close to the CPU, holding the most recently accessed data
- main memory (DRAM): services the caches and I/O
- virtual memory (disk): services the main memory

Access to virtual memory is thousands of times slower than access to main memory; computations requiring access virtual memory are basically infeasible.

Essence of software optimization over hardware

This leads to the concept of data locality: organize the data in such a way that what is needed is “close” most of time.

Some codes are called “cache aware” in the sense that they are tuned to a given hardware configuration.

A more useful concept is that of the *cache-oblivious* algorithms, which are meant to perform well on any system without knowing the details of the target architecture, thus greatly enhancing portability.

Cache-oblivious algorithms typically use divide-and-conquer techniques until every sub-problem created is small enough to fit into whatever memory hierarchy is being targeted.

A look to the future: exascale computing

Some predict that we will be in the exaflop era of computing by as early as 2018.

There are several technological hurdles that need to be overcome to reach exascale computing, most notably surrounding power consumption.

However, even after the hardware is built, there are still significant algorithmic issues that need to be resolved to realize the full potential of exascale computing.

Among these, we mention two that arise because of the sheer number of processes involved:

- a lack of global synchronization: the full exchange of information between processes will be infeasible

- a need for fault tolerance: there will be a virtual certainty of process failure during a computation

Summary

- real numbers, floating-point arithmetic
- numerical stability
- large-scale problems, parallelization
- rapidly changing requirements
- priorities