Chapter 22 -- Summary and conclusion

Section 22.1  Modeling and simulation in computational science and engineering

To summarize what we've seen:

1. Engineering design and scientific research uses a lot of mathematical modeling and simulation.
2. Developing or understanding the mathematical model is a significant part of the work.
3. Visualization (plotting, animations, etc.) can be key to understanding the situation described by the model.
4. While models can be quite useful and accurate, all models typically have only limited fidelity. Having a model does not necessarily mean that you automatically have something that will do a good job of approximately the physical situation. You could have made a mistake in specifying the model, or the mathematics used may be skipping important features of the situation.
5. Numerical work only approximates the mathematical situation described by a model. Even a perfect numerical program will introduce error due to the limited precision provided by floating point numbers. Useful deployment of a model always requires having some confidence that the error is small enough to have the calculations produce meaningful results.

These observations are true about modeling and simulation independent of what computer system is being used to do the calculations, so will be equally valid when using any other computer language or system.

Section 22.2  Basic tool-using skills

This course has covered features of Maple that are of benefit in personal use, such as:

1. Solution of equations, both exact and approximate.
2. Solution of simple optimization problems -- finding minima and maxima.
3. Algebraic manipulation, including symbolic differentiation, integration, and limit-taking.
4. High-precision calculation (using `evalf` with many more digits than the usual 10 or 15).
5. Plotting of formulae and of data onto graphs. The advantage is not that it's more convenient or better looking than a good charting tool such as Excel, but that it provides a way of both calculating the numbers and doing the plotting in one script. When doing engineering design and trying out lots of possibilities, the fact that both the calculations and the plotting can happen at the push of a button can increase productivity significantly.
7. Sorting tables of data. Some extra work may be needed for specialized sorting tasks, but it's easier than sorting large amounts of data by hand or writing a entire sorting program yourself.
8. Conversion of units, even compound units such as those used for acceleration or fluid flow.
9. Basic simulation: using models to approximate the state of a physical system at discrete points in time. In our work in this course, we used this information to create movies of what
Having such functionality on tap as conveniently as you'd use a handheld calculator is clearly useful for further engineering studies, as long as you expect to have to do some of the modeling and analysis yourself. You have been trained how to use Maple for this, but whatever you pick as your eventual "home base" system, you should find out how to do the same sort of things. You should know that whatever software package you pick ought to have such features available, and it should not be any more work to get them to happen than what you've seen with Maple.

## Section 22.3 Basic software development skills

Nowadays, there are many situations that can be handled by "software appliances" operated by point and click. However, if you get further into subjects that require the use of modeling and simulation, or need to extend your appliance to do additional things, you will need to return to the scripting and function-creating ideas covered here.

Key ideas include:

1. Selecting a system that has ample built-in expertise in the work domain you are interested in.
2. Assignment as a way of naming things and subsequently referring to computed results.
3. Evaluation of expressions -- the way things get done with computer languages. These express operations on symbols and numbers. The notational rules for entering expressions must be mastered in order to get much to happen. Fortunately expression syntax is highly similar between computer languages, so the effort in learning the rules for a second or subsequent language is typically not great.
4. Functions -- they are used not only in imitation of the uses they are put to in mathematics, but the input/output idea behind functions is the basis for building longer computations.
5. Scripting -- a sequence of actions that can be re-executed, possibly after a little editing. Eventually we moved from executing scripts by hitting return or enter for each line of the script, to placing the script in a code block and executing the entire block at once.
6. Data aggregations -- lists, sets, tables, etc. Adding together lists, using map, writing loops that operate on lists and tables are where the calculational power of the computer can be fully used.
7. Textual notation for computing: a programming language with its own rules for how to say things. Getting used to a typical declarative programming language such as Maple's is one of the biggest challenges in programming-- balancing parentheses and other delimiters, punctuation, special words such as for or if.
8. Writing the programming for new functions (procedures). We found that Maple, like most languages for technical work, modularizes scripts by turning them into procedures. This facilitates repeated invocation for testing, or for computational exploration.
9. Incremental development and frequent testing. In all the labs, a methodology of program development is used that involves building things one step at a time and executing a test on everything up to that step to see if it works. Only after the test is passed does work move onto building on the next step.
10. Documentation (program and script comments) saves time when returning to the work or letting other people use the work.
11. Consulting the on-line documentation to incrementally acquire knowledge of new features.

These ideas are the basis for work in most computer languages. When you need to come back to
this kind of work, you should recall the experiences with them in this course and embark on the process of re-acquainting and re-discovering these ideas in whatever language you settle into working on next.

**Section 22.4 Continuing an education in computing**

There are several different additional ideas that a full education in computational science and engineering should include:

1. More sophisticated models of physical situations often requires the use of *linear algebra*, and *differential equations*. Any of the technical systems (e.g. Maple, Matlab, Mathematica) can handle these calculations, either symbolically/exactly or with floating point calculation. It only awaits your becoming familiar enough with the mathematical ideas and taking the time to learn how the computational machinery you have already seen -- scripts, procedures, visualization, etc. -- is used with them.

2. More sophisticated calculations often require ideas from "advanced programming": *recursion, object-oriented programming, tree and network-graph data structures*. The history of computational science and engineering indicates that while for many decades mainstream applications avoided such features because of the additional effort needed by programmers to understand them, eventually the need for handling more sophisticated or complex situations has led to their extensive adaptation.

3. Most languages also support some form of *parallel computing* -- running parts of a program on several computer processors at once in hopes to getting results faster. For example, Maple can use all the processors of a "multi-core" computer such as is typical on many personal computers nowadays through a parallel version of *map*.

If you want to teach yourself about these things, how do you go about doing it? We have seen that most computer systems have ample amounts of documentation explaining their features. There are many books and web pages devoted to explaining these kinds of ideas. It takes a familiarity with fundamental concepts (which this course has exposed you to), time, and motivation. The latter is particularly important. We recommend having *a good excuse for learning the new material* -- a new problem that you want to solve, for example. Then it's a matter of reading, experimentation, and talking to knowledgeable and responsive consultants when you get really stuck.

As you get more experienced and ambitious, you should also become familiar with more of the principles of *software engineering*. As might be expected, once a programming project encounters success, it attracts many users who want to reuse the software. They put pressure on the developers to fix and extend the programming to do more. The techniques used to design and maintain a large system -- say one that has $10^5 - 10^7$ lines of programming, are far different from the informal and intuitive style we have used in this course. A program built for personal use -- taking a few hours to build -- is not at all like a large software system that takes dozens of programmers a few years to get working. Assuming that the "personal project" approach to software construction works on a large system makes about as much sense as assuming that the experience of building a backyard garden shed will be sufficient to let you build a 40 story skyscraper. Without organizational principles, a regimen of programming style, and software tools to do extensive testing and recordkeeping, it becomes infeasible to make even small changes to a large program without great expense. We've shown you a few software engineering ideas in this course -- frequent testing and incremental development, and the use of variable names that
are words rather than single mathematical symbols. The field of software engineering can provide you with the best practices and formal analysis taken from the construction of large computer systems over the past half century.