1

Why *numerical* software?

Numerical software is the software used to do computations with real numbers; that is, with numbers with decimal points in them like $\pi = 3.141\,5926\ldots$. These kinds computations are commonly of great scientific and engineering importance. Real numbers can be used to represent physical quantities (position, height, force, stress, viscosity, voltage, density, etc.). Computation with real numbers can be for simulating the nuclear processes in the centers of stars, finding the stresses in a large concrete and steel structure, or for determining how many spheres of unit radius can touch each other without penetrating. This kind of software is about *quantitative problems*. That is, the answers to our questions are not simple yes/no or red/green/blue answers. They involve continuously varying quantities. But computers can only store a finite number of values. So we have to use an approximation to real numbers called floating point numbers (described in Chapter 2).

Numerical software is often used for *large-scale problems*. That is, the number of quantities that need to be computed is often very large. This happens because we want to understand what is happening with a continuously varying quantity, such as stress in a structural column, or flow in a river. These are quantities that vary continuously with position, and perhaps with time as well. Since we cannot find or store the values at all infinitely many points in a column or a river, we must use some sort of ***discretization***. Discretizations are approximations to the true system, which are usually more accurate when more refined. Refining a discretization means that we create more quantities to compute. This does not have to go very far (especially for problems in three dimensions) before we are at the limit of current computers, including supercomputers.

How large are modern computational tasks? Here is an example. Consider water flowing through a pipe. If the flow is smooth, then we can use relatively coarse discretizations, and the scale of the simulation can be kept modest. But if we have turbulence, then the scale of the turbulent features of the flow are typically a fraction of a millimeter. To simulate turbulent flow in a pipe that is 5 cm in

diameter and 2.5 cm long with a discretization spacing of a tenth of a millimeter would involve at least $3 \times \pi(25\,\text{mm}/0.1\,\text{mm})^3 \approx 147$ million unknowns to store the flow's velocity. Just to store this would require over a gigabyte of memory (in double-precision). Unless you can store this in the main memory of a computer (the memory banks of your computer, but excluding disk drives), your algorithm is going to be slow. This amount of memory is simply the memory needed to store the results of the computations. The amount of memory needed for the other data used in the computation can be equally large or even much larger.

These large-scale problems are computational challenges that require not only effective and efficient algorithms but also implementations that maximize use of the underlying hardware.

1.1 Efficient kernels

Since we are trying to compute a large number of quantities, we need our software to be efficient. This means that the core operations have to be written to execute quickly on a computer. These core operations are often referred to as ***kernels***.

Since these kernels are executed many, many times in large-scale computations, it is especially important for them to run efficiently. Not only should the algorithms chosen be good, but they should also be implementated carefully to make the best use of the computer hardware.

Current Central Processing Units (CPUs) such as the Intel Pentium 4 chips have clock speeds of well over a GigaHertz (GHz). In one clock cycle of one nanosecond, light in a vacuum travels about 30 cm. This is Einstein's speed limit. For electrical signals traveling through wires, the speed is somewhat slower. So for a machine with a 3 GHz Pentium CPU, in one clock cycle, electrical signals can only travel about 10 cm. To do something as simple as getting a number from memory, we have to take into account the time it takes for the signal to go from the CPU to the memory chips, the time for the memory chips to find the right bit of memory (typically a pair of transistors), read the information, and then send it back to the CPU. The total time needed takes many clock cycles. If just getting a number from memory takes many clock cycles, why are we still increasing clock speeds?

To handle this situation, hardware designers include "cache" memory on the CPU. This cache is small, fast but expensive memory. If the item the CPU wants is already in the cache, then it only takes one or two clock cycles to fetch it and to start processing it. If it is not in the cache, then the cache will read in a short block of memory from the main memory, which holds the required data. This will take longer, but shouldn't happen so often. In fact, the cache idea is so good, that they don't just have one cache, they have two. If it isn't in the first (fastest) cache, then it looks in the second (not-as-fast-but-still-very-fast) cache, and if it isn't there it will

look for it in main memory. Newer designs add even more levels of cache. If you want to get the best performance out of your CPU, then you should design your code to make best use of this kind of hardware.

There is a general trend in the performance of these kinds of electronic components that is encapsulated in "Moore's Law", which was first put forward in 1965 [81], by Gordon Moore one of the founders of Intel:

the number of transistors on a CPU roughly doubles every eighteen months.

Gordon Moore actually said that the doubling happened every twelve months. But, averaging over progress from the 1950s to now, the number of transistors on a "chip" doubles about every eighteen months to two years. This usually means that the number of operations that can be done on a single CPU also roughly doubles every eighteen months to two years. But memory speed has not kept up, and data has to be transfered between the main memory and the CPU. To maintain the speed, we need to get memory and CPU closer; this is why we have multiple level caches on CPUs. It also means that if we want to get close to peak performance, we need to take this structure into account. There will be more on this in Part III.

1.2 Rapid change

As scientists, we are interested in research. That means that we want to go where no-one has gone before. It means that we want to investigate problems and approaches no-one else has thought of. We are unlikely to get something profoundly important on our first try. We will try something, see what happens, and then ask some new questions, and try to answer those. This means that our software is going to have to change as we have different problems to solve, and want to answer different questions. Our software will have to change quickly.

Rapidly changing software is a challenge. Each change to a piece of software has the chance to introduce bugs. Every time we change an assumption about what we are computing, we have an even bigger challenge to modify our software, since it is easy to build in bad or restrictive assumptions into our software.

The challenge of rapidly changing software is not unique to research, but it is particularly important here. In numerical analysis, a great deal of thought has gone into designing algorithms and the principles behind them. The algorithms themselves, though, have often been fairly straightforward. However, that has been changing. Consider Gaussian elimination or LU factorization for solving a linear system of equations. The standard dense LU factorization routine is fairly easy to write out in pseudo-code or in your favorite programming language. A great deal of analysis has gone into this algorithm regarding the size and character of the errors in the solution. But the algorithm itself is quite straightforward. Even when we add

pivoting techniques, there is much more to the analysis than the algorithm. But with the increasing importance of sparse matrices with general sparsity patterns, the algorithms become much more complicated. Current algorithms for solving sparse linear systems are supernodal multifrontal algorithms that use combinatorial data structures (elimination trees) that must be constructed using moderately sophisticated techniques from Computer Science. You can write a standard LU factorization routine with partial pivoting in an afternoon in almost any programming language. But current supernodal multifrontal algorithms require much more care to implement. This trend to increasing sophistication and complexity can also be seen in the development of finite element software, ordinary differential equation solvers, and other numerical software. When we start bringing together different pieces of software to solve larger problems, we should realize that we have a large software system, and it should be treated as such. Writing your own (from scratch) is no longer an option.

1.3 Large-scale problems

Problems in scientific computing usually involve large amounts of computation. These are called *large-scale problems* or *large-scale computations*. This can be because the problem requires a large amount of data (such as signal processing), produces a large amount of data (such as solving a partial differential equation), or is simply very complex (some global optimization problems are like this). Some tasks, such as weather forecasting, may both require and produce large amounts of data.

1.3.1 Problems with a lot of data

Signal processing is an area where vast amounts of data must be processed, often in real time – such as digital filtering of telephone signals, transforming digital video signals, or processing X-ray data in computerized tomography to get pictures of the inside of human bodies. Real-time constraints mean that the processing must be very rapid. Often specialized hardware is used to carry this out, and perhaps fixed-point rather than floating-point arithmetic must be used.

Other situations which do not have real-time constraints are seismic imaging, where pictures of the rock layers under the ground are obtained from recordings of sounds picked up by buried sensors. The amount of data involved is very large. It needs to be, in order to obtain detailed pictures of the structure of the rock layers.

In addition to the usual numerical and programming issues, there may be questions about how to store and retrieve such large data sets. Part of the answers to such questions may involve database systems. Database systems are outside the scope of this book, but if you need to know more, you might look at [26].

Many problems, such as finding the air flow around a Boeing 747, produce a great deal of data. After the relevant partial differential equations have been discretized, there are large systems of nonlinear equations that need to be solved. And the answer involves a great deal of data. Part of the problem here is making sense of the answer. Often the best way to describe the answer is to use computer visualization techniques to paint a picture of the airflow. Moving pictures can be rotated, shifted and moved enable aeronautical engineer to study different aspects of the problem. Computer visualization is outside the scope of this book, but if you need something like that, some good references are [37, 51, 68].

1.3.2 Hard problems

Sometimes, the data for the problem, and the data produced, are not very large, but solving the problem can involve an enormous amount of time. This can easily be the case in global optimization, where there can be very large numbers of local minima, but only one global minimum. Finding all the local minima and comparing them to find the best – the global minimum – can be like looking for a needle in a haystack. These problems are often closely related to combinatorial problems (problems which can be described in terms of integers rather than real numbers). Sometimes combinatorial methods can solve these problems efficiently, but often, continuous optimization methods are an essential part of the solution strategy.

Another area where there are hard problems which do not necessarily involve a lot of data are highly nonlinear problems. Highly nonlinear problems can have, or appear to have, many possible solutions. Perhaps only one or a few of the solutions is really useful. Again we have a problem of finding a “needle in a haystack”.