Chapter 3

. . . and how to solve them: A survey of techniques

This section will describe a general approach to solving quantitative problems and provide a quick survey of possible methods, resources and directions of attack.

Getting Started: Formulating the problem

1. Start with a physical problem that you are interested in.
2. Generate appropriate conservation equations.
3. Scale and approximate the equations until they are tractable.
4. Consider appropriate boundary and initial conditions
5. At this point you're problem will probably fall into one or more of several categories
   (a) Sets of Ordinary Differential Equations (ODE's) with a single independent variable
   (b) Initial Value problems (PDE's)
   (c) Boundary Value problems (PDE's)
6. Now figure out how to solve it

Fundamental Tools 1: Paper and Pencil

1. Find an exact analytical solution: If you're very lucky someone extremely clever (or yourself) will have worked out a solution to your problem (or one close enough for jazz). This is rare but always the best approach. If it's a classical problem (and most were worked out last century) it should exist in a book or journal (use Columbia's Library-Web `http://www.columbia.edu/cu/lweb/`).
2. Find an approximate analytical Solution: Often by setting coefficients to constants, considering small perturbations, or just throwing out terms *a priori*, your equations can be muscled into something that is straightforward (or someone has already solved). If the problem is essentially

linear these solutions will be quite good. If the problem is non-linear, BEWARE, stopping too soon can lead to dull solutions. Nevertheless, approximate solutions are very important for checking numerical solutions.

3. Produce a simple scaling argument: Very powerful technique that is usually correct to a factor of $2, \pi, 5$ or 10. You will have to do this at some point in the process, best to do it before you write a single line of code.

Fundamental Numerical Tools: algorithms and discretization If at this point you still need a numerical solution you will need to choose a method. All numerical solutions are discrete approximations to continuous solutions. Most of the differences in techniques arise from the choice of how to discretize the problem. Given a discrete set of linear or non-linear equations the issue then becomes how to solve them; however, the first step is choice of discretization. Some common methods and their pros and cons are.

1. Finite Differences: Approximate a function at a point by a truncated Taylor Series and combine the series of adjacent points to approximate the governing equations.

   pros Simple, efficient easy to code in 1, 2 and 3-D. Good for simple geometries, static meshes and problems with smoothly varying properties. Often the fastest way to get a feel for the problem.

   cons Not as good for complex geometries with sharp changes in material properties. Not particularly good for tracking internal boundaries.

2. Finite Elements: Approximate a global function as the sum of a set of local *shape functions* defined over a set of "elements" where the solution is approximated locally by some interpolating function. Link all the elements together and require that the error be a minimum in a global sense.

   pros Very good for problems with complex geometries, strongly varying internal properties or a need to track internal boundaries and materials. Also good for local refinement in resolution and Lagrangian moving-mesh problems.

   cons Can have significantly greater computational complexity on an unstructured mesh. Standard techniques can be expensive (computationally) to scale up to larger problems. Finite elements have just as many (or more) numerical artifacts as finite difference. In general, except for simple problems finite elements are less easy to code yourself although there are publicly and commercially available packages.

3. Finite Volumes/Finite Volume Elements: a hybrid between finite element and finite difference techniques. Discretizes space as a set of discrete volumes that trade fluxes at their boundaries and considers the

volume averages of properties within the volume. Finite Volume Elements also use some of the finite element formalism to define interpolation schemes between nodes. Finite Volumes is the natural extension of the conservation approach for coming up with discrete equations.

pros Combines the ease of finite difference with the formalism of finite elements. Is a useful approach for deriving discrete equations in stranger geometries and more general problems. Can be naturally conservative.

cons This technique has all of the numerical artifacts of any discretization. Also usually requires that material properties be smoothly varying over the mesh spacing.

4. Spectral and Pseudo-Spectral techniques: Rather than discretizing the functions in space, the functions are replaced by truncated Fourier series (discretized in frequency) or other orthogonal functions (e.g. Legendre Polynomials, spherical harmonics, Bessel Functions, Chebyshev polynomials. . .). Usually only the spatial terms are transformed to frequency, the time dependent terms are handled as ODE's or with finite difference techniques.

   pros Good for wave problems. Can be efficient if they make use of FFT's and orthogonal functions. If the problems are band width limited, these solutions are effectively continuous up to the highest frequencies maintained. Can also often reduce a PDE to a set of ODE's that can be solved efficiently (e.g. the Lorenz Equations).

   cons Usually only works easily for constant coefficients, regular geometries, simple boundary conditions and smooth functions.

5. Particle-based/characteristic methods/Semi-lagrangian methods: Uses much of the physics of hyperbolic systems to track particles around on a regular grid. Combines the best of random particle methods and Eulerian grid based methods. Examples include Semi-Lagrangian methods and "Contour Surgery" methods.

   pros Good for advection dominated problems. Low dispersion, proper upwind structure, can move relatively steep fronts through regular grids, has **NO** Courant stability condition so it can decouple time steps from spatial resolution. Excellent for open *outflow* boundaries One of my favorite schemes.

   cons The simplest form is not conservative nor preserves monotonicity or positivity (i.e. can introduce wiggles). Can be computationally expensive. . . requiring significant per point computation due to significant interpolation costs. Can be problematic around complex boundaries. Slightly more difficult to parallelize.

6. Microscopic techniques: In addition to standard "finite something" techniques which are used to solve macroscopic continuum equations. Some problems can also be solved by approximating microscopic principles. Several common approaches are "Lattice Gas cellular Automata",

molecular dynamics, and granular media which all approximate the motion of large sets of particles that are constrained either by "interaction rules" on a fixed grid or by "potentials".

pros Good for wave problems. Very good for large parallel machines. Can handle very complicated internal structures with simple rules.

cons Need large number of particles. Not a very good technique for static or quasi-static problems. In cellular automata it is sometimes quite difficult to relate the interaction rules to the macroscopic constitutive relations.

7. Other techniques: The above list is by no means exhaustive and new techniques and improvements on old techniques are being introduced everyday. Some more advanced techniques that we probably won't get too but look promising for certain classes of problems are "Fast Multipole methods" (i.e. greens functions) or "boundary integral methods" for complex fluid-solid interfaces in simple fluids. The Extended Finite Element Method (XFEM) for embedding discontinuities in regular meshes and "Level Set methods" for tracking complex internal boundaries (and other things) among others. In general, SIAM (www.siam.org) is a good place to start when looking for numerical techniques.

Scrounge for code When you have decided on a technique, it can be very worthwhile to look for already written pieces of source code to implement it (particularly for complex algorithms). These pieces should not be treated as black-boxes but understood and tested as if it were your own code. Nevertheless, the typos you save may be your own. There are several useful sources of code for many problems and many of them are free and available on the network.

Numerical Recipes `http://www.nr.com/` Modestly priced, well documented codes for just about everything but PDE's. Comes in most flavors of languages (FORTRAN (f77 and f90), C, C++). Probably one of the more accessible books on the subject but beware some of the codes are flaky (but improved in the second edition). Note: these codes are licensed and not public domain (or "open source" in the sense of the Gnu GPL). `.com` means commercial...

Netlib `http://www.netlib.org/` Netlib is an on-line library of public domain numerical codes stored at the University of Tennessee and Oak Ridge National laboratories. Netlib interface also provides access to High-performance computing software (HPC-Netlib), sources for parallel processing software, a high-performance computing data-base and other useful things.

PetSc the Portable, Extensible Toolkit for Scientific Computation. An enormous package (over 15Mb) of scientific codes for both serial and parallel computers written in an Object Oriented style modular format. Version 2.1.6 is currently available at `http://www.mcs.anl.gov/petsc/`

and has been installed on our current 64 processor linux cluster. This package provides a useful intermediate level for handling discrete systems of linear and non-linear equations in serial and parallel without requiring extensive low-level parallel code development. Includes a large host of iterative schemes including multi-grid. Written in C and Fortran although the C binding is perhaps the better of the two to use. The learning curve on this is quite steep but the web page provides a large number of examples. I hope to introduce it toward the end of this class. Could well be the shape of the future.

Commercial software There are several commercially available software libraries, most Notable are IMSL, NAG and IBM's ESSL. These are very high quality, extremely efficient, tested routines but you can't access the source code. Useful for production work but not portable.

Other sources: the library Never hurts to see if someone has already done your problem. The engineering library at Columbia has quite a large numerical methods collection. When you have a better idea of what you are looking for use Columbia's LibraryWeb to go find it anywhere. Also much of the numerical literature can be found on-line (e.g. the SIAM journals).

A note on Languages You can program in pretty much any language you please, however much of the classic source code in earth sciences is written in FORTRAN (both f77 and f90) and to a lesser extent in C (although this is changing rapidly with a particular increase in C++). These are currently the two principle *compiled* languages of scientific computing. Unfortunately, at this very moment, advice on which way to go is rather in flux. For portability, the gnu C compiler (gcc or g++) is perhaps the most commonly available, open-source and free compiler which supports most of the open-source world. Fortran support under gnu is not as good. There is a free f77 compiler (g77) and an attempt to support f90/f90 in the g95 project (`http://g95.sourceforge.net`) although as of Feb. 2004, the project is still defined to be in it's "pupal stage." For serious compilation in Fortran, a commercial compiler is probably necessary and there are several choices (that change daily) depending on platform. For simple array manipulation, FORTRAN is simpler, often has more efficient optimizing compilers and somewhat more flexible array handling in subroutines. C is better for complex data structures and string handling. However, the compilers are converging and new languages such as Fortran90, or C++ are rapidly taking hold. In addition to compiled languages, however, there are several interpreted languages that are very useful for scientific computing. Matlab (which of course is highly proprietary) and Python as a scripting language seem to be useful choices. However, according to everyone I've spoken to in the scientific computing arena, JAVA is a dog and will never be a serious language for scientific computing.

Put it together

1. Pick a platform: There is almost a continuous array of machines in terms of price, performance and ease of use (the latter is often inversely proportional to the first two). There are only a few things to keep in mind when picking the right machine for the job
   (a) You will often spend more time in development and testing than in running so finding a good programming environment is often more important than finding a fast box (although to be honest, the difference between what we develop on and what we compute on is becoming indistinguishable up to processor speed).
   (b) The biggest jump in style of programming will be between serial codes and parallel codes. Most of this course will emphasize serial computing in which case the performance of most fast scalar processes are comparable and the basic flavor of OS (Unix) and compilers (gcc) are qualitatively the same (but the devil's in the details). The jump to parallel computation is more difficult but coding environments like PETSc blur and ease the distinction.
   (c) Because things are changing so quickly, it pays to keep your code flexible, conservative and portable!

   Useful and available machines at Lamont are various flavors of Unix boxes including Sun "Workstations" (no longer recommended), Intel boxes running Linux (in N sub-flavors) and Macs running MacOS X (which is a real Unix kernel). Also available are several linux clusters including including our 64 processor Linux-Cluster Fats and the AQF cluster downtown. For bigger machines (CRAY's and really large clusters (e.g. 1000's of processors) you need to go out of house (but if you can't get going with what we have here, you have a seriously ugly problem).

2. The development cycle: Compile, Debug, Run (rinse and repeat ad nauseum). There are several tools that are useful for shortening the development time of codes.

   Development Environment While everything can be done with simple text-editors. `emacs` or `Xemacs` provides a powerful (and free) development environment for writing code. It understands many keywords, keeps your indentation clean and allows you to interact with the compilers and debuggers in a much more natural way. I tend to use Xemacs because it has a slightly nicer interface but all Emacsen are available on all platforms.

   Numerical debugger A good interactive debugger can be invaluable as it allows you to step through the code and see how it works and where it blows up. On the Suns the `Workshop` debugger was very nice but isn't portable. On most platforms Xemacs will interact (crudely) with GDB (the gnu debugger) and this is a portable if somewhat clunky approach. On the linux boxes using the PGI compilers they supply (a flaky) debugger called xpgdbg. And macOS X has a rather sophisticated development environment called XCode.

Visual debuggers A picture is often (literally) worth a million numbers. So a quick look at the output visually can often lead to spotting bugs much faster. For visualization of output in general try..

1-D problems Matlab (available on all platforms).

2-D problems Unix: Matlab, GMT system. various imagetools (pbmtools, xv, xanim for animations,gimp).

3-D problems Unix: Matlab (v6 and above), OpenDX (an open source data-flow 3-D visualization package based on IBM's DataExplorer).

In general we will do most of our work in Matlab which is great but not generally available without moolah.

3. Production: When it works and is bug free (or close enough for jazz) you still need to treat your original problem as a Numerical experiment and explore *parameter space* which is spanned by the values of your dimensionless parameters. The more you understand the less time you will waste in production!

4. Understanding: Once you have a full numerical description of the behavior of your numerical system you still need to accomplish two important tasks.

   (a) Prove to yourself (and others) that the codes are doing what they should be doing.

   (b) Reduce the numerical results to a simple parameterization that can be used by anyone without typing a single character.

   SCALING ARGUMENTS are your best bet for both of these tasks so it's back to paper and pencil.

5. Completion and writing up: When you understand your problem so well that you don't have to model it anymore it's time to write it up. Only at this point should you discuss the relevance of the model to reality. A good solution stands on its own and should be discussed on its own (you never can be wrong this way). Whether it is a good model of reality depends on the available data. Nevertheless, where this approach is most useful is when there is a minimum of data and you need some quantitative conception of what to look for. This is the most useful contribution of modeling.