6

Building Programs and Conclusions

6.1 Numerical libraries

Pre-written code is available for common tasks.

Code Repositories:

- *NETLIB* at www.netlib.org offers free sources from a myriad of authors.
- *Numerical Recipes*, http://numerical.recipes/, explains and provides a broad and selective collection of reliable subroutines. Each program is available in several languages: C, C++, Fortran 77, and Fortran 90. Code is proprietary but you will get the source code.
- The *Gnu Scientific Library* is a collection of open-source routines, www.gnu.org/software/gsl. It is written in C.
- *IMSL (International Mathematical and Statistical Library)* – proprietary
- *NAG (Numerical Algorithms Group)* Library – proprietary
- *SciPy* includes an open source library for Python.

The *Guide to Available Mathematical Software* http://math.nist.gov maintains a directory of subroutines from numerous public and proprietary repositories.

A few noteworthy specialized numerical libraries:

- A specialized, refereed set of routines is available to the public from the *Collected Algorithms of the ACM* at http://calgo.acm.org.
- An exceptional implementation is FFT-W, the "Fastest Fourier Transform in the West". It is a portable source code that first detects what hardware it is running on and chooses different computational strategies depending on the specific hardware architecture. This way, it can simultaneously achieve portability and efficiency.
- BLAS and LAPACK are highly optimized libraries for numerical linear algebra
- Computational Geometry Algorithms Library, http://www.cgal.org
- Stony Brook Algorithm Repository, http://www3.cs.stonybrook.edu/~algorith/
- PETSc is a framework for solving partial differential equations, http://www.mcs.anl.gov/petsc/

6.2 Programming

"In computer programming, the technique of choice is not necessarily the most efficient, or elegant, or fastest executing one. Instead, it may be the one that is quick to implement, general, and easy to check."

Numerical Recipes

To a large extent the same advice applies for scientific programming as for programming in general. Programs should be clear, robust, general. Only when performance is critical, and only in the parts where it is critical, should one compromise these principles. Most programmers find that it is more efficient to write a part, test it, and then code the next part, rather than to write the whole program first and then start testing. In software engineering this is known as "unit testing".

In other aspects, writing programs for scientific research is different from software development. Research programs keep changing and are usually used only by the programmer herself or a small group of users.

Most programs written for everyday scientific computing are used for a limited time only ("throw-away codes"). Under these circumstances there is little reason to extract the last margin of efficiency, create nice user interfaces, write extensive documentation, or implement complex algorithms. Any of that can actually be counterproductive, as it consumes time and reduces flexibility. Better is the enemy of good.

Debugging. It is easy to miss a mistake or an inconsistency within many lines of code. In principle, already one wrong symbol in the program can invalidate the result. And in practice, it sometimes does. Program validation is a practical necessity. Some go so far as to add "blunders" to the list of common types of error in numerical calculations: roundoff, approximation errors, statistical errors, and blunders. Absence of obvious contradictions is not a sufficient standard of checking. Catching a mistake later or not at all may lead to a huge waste of effort, and this risk can be reduced by spending time on checking early on. One will want to compare with analytically known solutions, including trivial solutions. Finding good test cases can be a considerable effort on its own.

Programs undergo evolution. As time passes improvements are made on a program, bugs fixed, and the program matures. For time-intensive computations, it is thus never a good idea to make long runs right away. Moreover, the experience of analyzing the result of a short run might change which and in what form data are output. Simulations shorter than one minute allow for interactive improvements, and thus a rapid development cycle. Besides the obvious difference in cumulative computation time (a series of minute-long calculations is much shorter than a series of hour-long calculations), the mind has to sluggishly refocus after a lengthy gap.

For lengthy runs one may wish to know how far the program has proceeded. While a program is executing its output is not immediately written to disk, because this may slow it down. This has the disadvantage that recent output is unavailable. Prompt output can be enforced by closing the file and then reopening the file before further output occurs.

Build automation. For fully compiled languages and programs that consist of many source and/or header files, when one file is changed dependent files must be recompiled. On Unix environments, `makefiles` help to build (compile and link) software by keeping track of the last time files were updated and only updates those files which are required. Type make and the commands in the makefile will be executed.

When data sets accumulate, it is practical to keep a log of the various simulations and their specifications. The situation is analogous to an experimentalist who maintains a laboratory notebook. Forgetting to write down one parameter might necessitate repetition of the experiment. And not labeling a sample can, under unfortunate circumstances, make it useless. Ironically, some scientists keep notes of their model computations not in an electronic file but in a real paper notebook.

For large or collaborative projects, version control systems are commonly used. These record and archive changes to files. Examples are *CVS* (Concurrent Versions System), *Git*, *Mercurial*, and *SVN*. Version control can also serve as backup and, if hosted on a website, is a mechanism for distributing code to users.

Data can be either evaluated as they are computed (run-time evaluation) or stored and evaluated afterwards (post-evaluation). Post-evaluation allows changes and flexibility in the evaluation process, without having to repeat the run. Numbers can be calculated much faster than they can be written to any kind of output; it is easy to calculate far more than can be stored. For this reason, output is selective.

To be compelling, a numerical result, or a conclusion derived from numerics, needs to be reliable and robust. This expectation is no different from any other branch of scientific inquiry; a laboratory measurement, an astronomical observation, and a theoretical formula are all expected to hold up to scrutiny, because scientists fool themselves often enough, but for numerics the standards are less established. Robustness and convergence tests go a long way. We can change a parameter that should not matter to see if the we get the same result, and change an input parameter that should matter to see if the expected change occurs. Fitting a graph to data ought to be robust, i.e. when a few of the points are removed the fit parameters should barely change.

6.3 Modern curve fitting

Fitting straight lines by the least-square method is straightforward. For linear regression we minimize the quadratic deviations $E = \sum_i (y_i - a - bx_i)^2$, where x_i and y_i are the data and a and b are, respectively, the intercept and slope of a straight line. The extremum conditions $\partial E/\partial a = 0$ and $\partial E/\partial b = 0$ lead to the linear equations $\sum_i (y_i - a - bx_i) = 0$ and $\sum_i x_i(y_i - a - bx_i) = 0$, which can be explicitly solved for a and b. The popularity of linear regression is partially due to do the computational convenience the fit parameters can be obtained.

Minimizing the sum of the square deviations yields the most likely fit for Gaussian distributed errors. The maximum likelihood fit maximizes the probability $P = \Pi_i p_i$ where $p_i = p(y(x_i; a, b, ...) - y_i)$. Using the product rule for differentiation, an extremum with respect to a occurs when $\partial P/\partial a = P \sum_i (\partial p_i/\partial a)/p_i = 0$, and hence $\sum_i \partial \ln p_i/\partial a = \partial(\sum_i \ln p_i)/\partial a = 0$. There is one such equation for each parameter. In other words, $\Pi_i p_i$ is maximized when $\sum_i \ln p_i$ is maximized. When p is Gaussian, the most likely parameters are those that minimize the sum of the square deviations.

Some other functions can be reduced to linear regression, e.g., $y^2 = \exp(x)$. If errors are distributed Gaussian then linear regression finds the most likely fit, but a transformation of variables spoils this property. If the fitting function cannot be reduced to linear regression it is necessary to minimize the error as a function of the fit parameter(s) nonlinearly. This is numerically far more challenging than linear regression, because, similar to nonlinear root-finding, one cannot be sure the optimum found is global rather than only local.

Fits with quadratically weighted deviations are not particularly robust, since an outlying data point can affect it significantly. Weighting proportional with distance, for instance, improves robustness. In this case, we seek to minimize $\sum_i |y_i - a - bx_i|$, where, again, x_i and y_i are the data and a and b are, respectively, the intercept and slope of a straight line. Differentiation with respect to a and b yields the following two conditions: $\sum_i \mathrm{sgn}(y_i - a - bx_i) = 0$ and $\sum_i x_i \mathrm{sgn}(y_i - a - bx_i) = 0$. Unlike for the case where the square of the deviations is minimized, these equations are not linear in a and b, because of the sign function sgn. (They happen to be still easier to solve than by nonlinear root finding in two variables. The first condition says that the number of positive elements must be equal to the number of negative elements, and hence a is the median among the set of numbers $y_i - bx_i$. Since a is determined as a function of b in this relatively simple way, the remaining problem is to solve the second condition, which requires nonlinear root-finding in only one variable.)

Another type of error analysis made possible by numerical computations is error determination by randomization. The uncertainty is determined based on a large number of random deviations in the input parameters. This is especially useful for discrete or discontinuous problems where error propagation using derivatives is not possible.