Using UNIX for Science

Paul Gettings
gettings@mines.utah.edu

April 5, 2004

1 Introduction

UNIX is an old operating system (originally from the early 1970s) which has been continually updated and extended to the present day. Despite this constant evolution, the basic interfaces, libraries, and commands remain nearly unchanged after decades. This stability of commands, coupled with the inherent stability of the OS, makes UNIX (in all its' variants) attractive to the scientist needing to do computer work.

Scientists are paid to do science, not program computers or fight with system administration tasks. Hence, the stability of the UNIX OS means that a scientist need only setup a machine once every few years (or decades). With automated update tools, such as the apt-get utility of Debian (and other) Linux, the system can effectively maintain itself with daily software upgrades for security issues, etc. Since the library and command interfaces are almost totally static, software written to the UNIX APIs from the 1970s still works without change on a modern machine and OS; the scientist can spend time doing science, not porting or reimplementing old code.

Extensions to the UNIX infrastructure can be utilized if desired or necessary, but this code can be separated from existing tools and code, allowing a scientist to write only as much code as is absolutely necessary. It is not necessary to revise all the existing tools and software for each revision of the OS. Again, the scientist can spend more time doing science, and less fighting with the computer.

1.1 Target Audience

This document is designed for a scientist who is new to UNIX (although perhaps very experienced with Windows or something else), but knows the basic commands to log in, start a shell, navigate a directory structure, etc. Users with experience in other environments (Windows, MacOS, etc.) are encouraged to compare what is described here with their old environments, and tweak as they desire.

This document is meant to introduce various bits of "wisdom" gained through the use of UNIX as a computing environment for science. It does not attempt to teach UNIX, the philosophy of UNIX (see *The Art of Unix Programming* by Eric S. Raymond), the history of UNIX, or the suitability of UNIX for any other purpose (e.g. secretaries).

Readers with experience in UNIX or any other environment have likely already formed patterns of work which may or may not surpass those presented here. These ideas are not laws, commandments, or strong suggestions. They are the distilled elements of a system of work evolved through trial, error, and much rewriting. Ideally, a new UNIX user doing science will read this document, see how to apply these ideas, and skip the long, tortuous, struggle to create their own working scheme.

Comments, suggestions, and corrections should be sent to the author. Email contact is preferred; anonymous email is fine. The author's email address is `gettings@mines.utah.edu`.

2 Directory Structure

UNIX is based on storing data in bits of the disk, labeled as `files`. Files can be nearly any length, although modern OSs impose a limit of between 2 GB and 64 TB (kB=1024 bytes, MB=1024 kB, GB=1024 MB, TB=1024 GB) on file length (depending on whether the file offset counters are 32- or 64-bit integers). Files are often small chunks of data, and UNIX has filesystems built to insure efficient storage of small and large files, and is *very* good at insuring files are not fragmented across a disk. This is in distinct contrast to the older Windows filesystems, which would fragment files quite heavily, slowing file I/O considerably. Users used to the old (pre-NTFS) Windows world should note that UNIX does not *ever* require a defragmentation of a disk.

2.1 Files and Directories in UNIX

Files in UNIX can have names that are less than 1024 characters long. There is no absolutely forbidden character in a name, other than (possibly) "/", which is used as the directory separator. Many characters (?, *, !, etc.) are more difficult to use than standard letters and numbers, but they can be used. As noted before, files can be anywhere from 0 bytes to 64 TB in size, although most files are around 1 kB in size.

Directories have the same naming rules as files, and are effectively just a small, special type of file. They hold the entries for files and directories that are contained below them in the *directory tree.* All UNIX machines have a directory tree that starts at the root directory "/". Users generally have a home directory located under `/home`.

UNIX machines do not have a concept of drive letters or names. Instead, all disks on a system (including network-accessible disks) are given a unique *mount point*, or directory, where they are accessible. A disk is mounted to a directory, and the contents appear as files and directories under the mount point. Thus, a typical UNIX server will have one disk (or partition) mounted for `/`, another for `/usr`, another for `/home`, and perhaps also one for `/var`. To a user, all these disks appear as a single, coherent, filesystem. This means that an administrator needs to keep track of what is mounted where (for space concerns), but a user need not. If a disk fills, an administrator can move all or part of the data on that disk to a new disk (under a new or same mount point), and the user will not notice except that the space available increases.

This ability to make disks and the filesystem independant, to maintain constant absolute paths to a given file regardless of the disk it is actually stored on, is invaluable to a scientist. Upgrades of the disk space on a UNIX machine are not accompanied by a great reorganization of file locations; the disks simply get bigger, or entire directory trees are shifted to new disks, while retaining their old filesystem locations! Hence, once an absolute path is chosen, it can be maintained for all time. This makes maintaining software and data heirarchies vastly easier.

2.1.1 Standardized Directory Names

UNIX, over its' long history, has standardized some directory names and file locations across all implementations. The following describes some of the "special" names. These are not hard and fast rules, but are strongly followed customs among UNIX administrators and software designers.

1. **/etc** holds system-wide configuration files, often in subdirectories.
2. **/usr** holds system binaries and data for users, such as editors, X11, graphing programs, and so forth. **/usr** has many subdirectories, including **/usr/bin** for the binaries, **/usr/X11** for data and programs for X11, **/usr/etc** for special configuration files, and **/usr/local** for local software not part of the standard UNIX distribution.
3. **/var** holds log files, cache directories, and spool directories for mail and printing. **/var/tmp** is also used by some editors for temporary storage, as **/var** is often on a different disk than **/tmp**
4. **/tmp** is meant for temporary storage by all users. This is often its' own disk (or partition), so filling /tmp does not impact the rest of the system.
5. **/bin** and **/sbin** hold system binaries. These are different from the programs in /usr/bin in that they are designed as the basic programs to run the system. Programs in /bin and /sbin include the user shells, system startup and shutdown, and run-time library handlers.
6. **/lib** and **/usr/lib** hold system libraries. These are chunks of code used by many programs to do common tasks. Rather than keep a copy in every program, the code is kept once in the library, and the run-time dynamic linker stitches the program and libraries together. /lib holds libraries for the system binaries, /usr/lib for the binaries in /usr/bin.

7. **/home** is the typical location of user home directories. Ordinary users in UNIX are not allowed write permission to the system directories, so they need a place to store their own files. These go in their "home directory", normally /home/`username`. Within the home directory, a user normally has absolute authority, although the system superuser (root) can modify files if necessary.

These directories are present on virtually all UNIX systems, although a specific file may change its' location depending on the UNIX distribution and system administrator. Often, administrators who move files from the "standard" locations will use a symbolic link to help users navigate the directory tree.

2.2 One Directory per Project

Individual users on UNIX can generally only write to their own home directory, and subdirectories thereof. This means that all of a user's files are normally in a single directory structure rooted at /home/username. If a user creates all files in their home directory, it will quickly become cluttered with thousands of files, which will need increasingly complex names to avoid `namespace collision` (only one file can exist with a given name in a directory).

Therefore, it is best to give each scientific project its' own directory, under the home directory. All data, results, and interpretations can be kept in the project directory. Programs which are used by many projects can be kept in a "bin" directory in the home directory, and referenced easily as "~/bin/`program name`" in scripts and documentation.

When each project has its' own directory, namespace collision is minimized, so results and data don't get overwritten on accident. Each project directory can also be the root of its' own standardized directory tree, with (for example) a subdirectory `doc/` for reports and documents related to the project, `data/` for raw data files, `processed/` for processed data, etc. If each project directory has roughly the same structure under it, it is easier to find results, documents, and data years after finishing a project.

Finally, if each project has its' own directory, it is easy to backup or move an entire project using UNIX file tools; package the entire directory into an archive (see `tar` or `cpio`) and transfer the one archive to another location. This makes collabaration significantly easier.

2.3 Keep Data and Results Together

Keep the raw data for a project together with the results and interpretations. This makes it easier to move and archive a project, and also makes it easier to discover what was done after a project is finished.

If the raw data and results are in the same directory tree (under the project directory), then scripts which operate on the data and results need short relative paths, which are less prone to breaking (less fragile) than long, absolute paths. Scripts can use a shallow, local directory structure in the project directory to track processing steps. This structure can be specialized to the project (or subproblem), without worrying about breaking another project's scripts.

2.4 Symbolic Links Make Life Better

If it is necessary to have final results from multiple projects in a single directory off the home, use symbolic links to link the final results from the project directories to somewhere else. Symbolic links explicitly show the link target, so the source of the results can be quickly determined.

If files from one project are needed in another, use symbolic links rather than copying the files. Copied files must be updated by hand; symbolic links need no updating.

Raw data files should generally be kept in a primary location for use, and also in a backup location in case of corruption or accidental deletion. In this case, do *not* use links, symbolic or otherwise.

3 Data Files

3.1 Disk is Cheap, Data is not

Raw data is the most precious of all scientific commodities; once lost, it can never be regained. Therefore, it is worth the disk space to keep all original data, including that from experiments which didn't quite work. Since disk space is already cheap, and only gets cheaper, it is also worthwhile to keep processed versions of the data in addition to the raw data. In no case should the original data be destroyed in favor of the processed results. Processing represents computing time, which is not work; raw data represents real work.

Processed data which is particularly large and that can be automatically regenerated can be discarded to save disk space, if necessary. Always assume the raw data is more correct than the processed data - that is, if in doubt, reprocess the raw data rather than rely on the processed results. This is a fundamental reason to make data processing an automated process using scripts and programs.

Raw data often requires some quality checking to remove points which are know to be spurious or irrelevant. This should be done from a copy of the original file. Keep the untouched original in a separate directory (and preferably in an offline backup as well). Keep detailed notes of what was changed between the original and checked versions, and keep those notes in a file with the data. Plain-text files are easy to make in UNIX; use vi, emacs, or any other text editor. The format is not important, so long as the notes are human-readable, clear, and explicit about the changes and the reasons for the changes.

This file of notes will be important when interpretation is done of the results; questions about the data will need to be answered from the notes in this file. Make sure the notes are not lost, are kept with the data files, and are kept up-to-date. If there is a need to recreate the quality-checked data file (e.g. disk crash, accidental deletion), the notes will be the algorithm used to do this. If the notes are not complete, do not explain why a change is made, or how the changes are made, the scientist will need to rederive everything from scratch. This is expensive; disk is cheap.

3.2 Text is King

Given the option, data should be stored in a plain-text, human-readable format. This means a human, looking at the file with a pager or editor, can read and understand the numbers/strings/etc. without a massive translation table in their head. Proprietary formats do not normally fall into this category. Also note that many modern text formats, such as XML, do not fall into this format.

Scientific data, which is typically giant sets of numbers, does not need the expressiveness and self-documenting features of XML, SGML, or a relational database. All that is necessary is a well-delimited table of numbers, preferably with a header record to detail what columns represent.

3.2.1 Plain-Text is Readable

Good data formats are plain-text, allow for arbitrary comment records (generally they start with a # or % character), and do not depend on fixed column indicators for field separation (like the old Fortran formats). Whitespace, commas, or colons are common delimeters in data formats. Ideally, lines should not be more than 150 characters long.

The benefit of format conforming to these rules is that a human can directly read and edit the data file using only a text editor. Edits to the data file can be commented in the file itself, making it difficult to lose the editing comments without losing the data.

3.2.2 Plain-Text is Self-Documenting

Plain-text formats are effectively self-documenting, assuming there is a header record for the table. There is not generally a need for a program to automatically determine the format and semantics of a scientific data format. It is generally necessary for a human to be able to determine the format and meaning of a data file. Humans work easier with a table with headings than with the explicit tagging used in machine-friendly formats such as XML.

Numbers stored as text have three additional useful properties: (1) they do not worry about endian issues (see section 3.3), (2) text files compress very well using `gzip` and similar schemes, and (3) text is easy to process with standard UNIX tools (e.g. sed, awk, grep, perl).

3.2.3 Plain-Text can use Text Tools

Data files stored as delimited text are easy to process using standard UNIX tools, which were designed to process streams of plain text. Programs such as `awk` and `bash` can do complex record-oriented processing, including floating-point math. Since the utilities were designed to act as filters, multiple programs can be stitched together using shell pipelines. The result is a specialized processing algorithm that can take a raw data file and process to a final result using nothing but common tools in a self-documenting format.

Once stored in a script, such a pipeline becomes a single, specialized command that can itself become part of a pipeline or processing flow, with the details hidden until needed. The script documents the procedures for future reference (such a three years after a project ends) or modification.

Since GNU tools are available for all UNIX platforms, processing pipelines can be built to a single set of commands, and run anywhere with generally no porting issues. Although these shell pipelines will be slower than a custom-built processing code, generating the custom code will generally take many orders of magnitude longer than the shell pipeline. For all but the largest processing jobs, the shell pipelines will be more than fast enough. Again, computing is not real work; programming is.

3.3 Binary Formats have a Place

Some data sets are too large to efficiently deal with as text files. For these (few) cases, a binary format is the only practical option. These problems deal with data sets that are multiple GB in *binary* format; the equivalent uncompressed text format would be tens of GB on disk. On current top-end machines, with 64-bit disk addressing, the practical limit for moving to a binary format is closer to 100 GB or more on disk.

Binary formats are typically 3-10 times more compact than a text format. The penalty is that binary formats require special tools to view, edit, and reformat. Standard tools do not handle binary format well, and hence the scientist will generally need to construct a custom-built code in C, Fortran, Java, or similar language to read, write, and edit the data set.

Clearly, binary formats are to be avoided if possible, but some data sets are simply too large (or were, at the time of creation) for text formats. When a binary format is to be used, it should still be kept simple. Preferably, the data set should have two files: (1) an ordered stream of values - no record delimiters, strings, or headers; and (2) a header file in plain text that holds the header information, data set layout, binary value encoding (endianness, byte-count per value, etc.).

This scheme will allow a future researcher (perhaps even the creator years after designing the format) to (re)create tools to deal with the format. This retains most of the self-documenting features of a text format, while keeping the large data set easiest to alter.

Embedded strings should be avoided in binary formats. If the binary file must be shift between machines of different endianness (see below), embedded strings will make the translation vastly more complicated.

3.3.1 The Curse of Endianness

Different machine architectures store binary values in different ways. In particular, the bit patterns for floating point and integer numbers is uniquely implemented (by the IEEE specification), but the *order* of bytes in memory changes. Since a binary file is ordinarily written as a direct dump of a memory location, bit for bit, in the order memory is stored, the different ordering schemes leads to the problem of *endianness.*

Sun, Motorola, and other RISC CPUs use an order where the most significant byte appears first in memory. This is known as *big endian.* Intel, AMD, and other x86 chips use the opposite order, where the least significant byte is stored first. This is known as *little endian.* If a binary file is created on a big endian machine, and read on a little endian, the resulting values will be wildly wrong.

Transforming between big and little endian can be done, and is most easily done for 2-byte data types (16-bit integers). In that case, the standard UNIX utility `dd` can do the byte swapping; see the manual for exact syntax. For floating point values (using 4 or 8 bytes per number), the procedure is more involved, and there is not a standard UNIX command to do the swapping. Instead, a custom tool must be built in C or similar language. Fortran is difficult to use for this task, due to the difficulty in direct memory access.

If a binary format has varying-length entries (e.g. 16-bit integers followed by 32-bit floating point numbers), the tool must be more clever yet, and read each value into memory, swap it according to its' length, and then write the result to a new file. If strings or other variable-length fields are also embedded, the resulting tool can be very complex.

Endian issues are the major reason binary formats should not have embedded strings, header records, or other information. The easiest byte-swapping tools are those which can assume all entries are a single fixed length.

4 Programs

Programming is an important part of modern science, as the ability to process large amounts of data becomes more necessary. However, scientists are paid to research science, not produce code. To a scientist, then, programming is a tool in service of the goal of expanding science, not an end in itself. The less time spent programming, the more time can be spent doing science.

Most scientists, since they are paid to do science, are poor programmers (most, not all). The code scientists write normally works, but it is often fragile and difficult (to impossible) to maintain. Scientific training often does not include much, if any, training in how to write good, maintainable code. Even those scientists who do write good code often end up reinventing an existing algorithm to solve a particular problem. The scientist's solution, being the product of a single attempt, will often work well enough for the problem at hand, but it is normally inefficient, fragile, and complex compared to the more refined algorithms developed by computer programmers over the years.

The problem of reinventing poor versions of well-known algorithms is best solved by scientists having access to proficient (professional) computer programmers, who have been trained to generate good, maintainable code. These programmers will be able to see the science problem as a case of a general algorithm (often developed for something completely different), and then know where to find an efficient, tested, implementation of a suitable algorithm. Most scientists will not have access to any type of programmer most (or all) of the time, so a stopgap is to find ways to improve the coding of the scientists, so they can produce more maintainable code.

4.1 Interpreted Languages are Nice

Interpreted programming languages, such as Perl, Python, or Java, are a good beginning to helping scientists produce maintainable, readable, code. Interpreted languages are high-level, and come with large libraries of functions to handle file and user I/O, text processing, and mathematics. These library functions are efficient, and well-documented, so another researcher can look at code and decipher the algorithm being used. This is important for collabarative work as well as porting old software to new machines.

Interpreted language syntax, once learned, is generally more expressive than compiled languages (e.g. C or Fortran), although it is certainly possible to create arcane programs in any language (except, perhaps, COBOL). Intepreted languages encourage more transparent algorithm implementations, through their more expressive syntax. Compare a simple program in Perl, Python, and C++ to see an example of this.

Interpreted languages are also generally much faster to code and debug. Languages such as Perl and Python have built-in garbage collection, freeing the scientist from worrying about dynamic memory allocation and memory leaks. The languages can enforce bounds checking on arrays, and intelligently handle exceptions without any programmer instructions. The data types are often more rich (e.g. associative arrays in Perl, hash tables in Python or Java), which greatly simplifies implementation of many algorithms.

The cost for faster generation and debugging is slower execution speed. Interpreted languages (Java is an exception here) are slower than the same algorithm implemented (correctly and efficiently) in a compiled language. The speed loss is generally not great; large computation loops in Python or Matlab are notably slower than compiled versions, however. Suitably recasting the problem (e.g. translating computation loops into matrix operations for Python and Matlab) can often keep the speed loss of interpreted languages to no more than 50% (in the worst cases).

Half the speed of compiled languages sounds like a large loss; one that should be avoided by using compiled languages. However, there are two points to keep in mind: (1) computations are generally "small" - the difference between 50 microseconds and 100 microseconds is irrelevant; and (2) programming interpreted languages is vastly easier than compiled, often a factor of 10 or more in terms of time spent programming and debugging. Except for the largest problems (e.g. global circulation models, globular cluster simulations, or stellar cores) interpreted languages are more than fast enough to yield practical run times.

4.2 Compiled Languages are Fast

Some problems are big enough that interpreted languages are too slow for practical use; computations in Perl will take hours vs. minutes if done in C. Some algorithms cannot be reworked from loops and functions to matrix and vector operations, so numerical computing environments (e.g. Numeric Python, Mathematica, or Matlab) cannot be efficiently used. These (few) problems need the speed of a compiled language.

Compiled languages are fast; as fast as can be run on the hardware. They are much harder to write, have smaller built-in libraries, and take much more time to debug than interpreted languages. Scientists who write compiled language code generally write arcane, nearly unmaintainable code that is very fragile. Few scientists have the experience, training, and desire to write good, maintainable, compiled code. This is because scientists are paid to do science, not programming.

4.2.1 Choice of Language

Once it is decided that a compiled language is necessary, the choice of what compiled language to use must be made. This is often made based on what languages the scientist knows, and nothing more. All compiled languages make "fast" code, but some make faster code than others.

Fortran still produces the fastest numerical code possible. Given equally tuned compilers for a platform, the Fortran compiler can produce faster code than a C compiler. This is due to language design differences, and was a significant factor in the continuing use of Fortran for new numerical codes. With the increasing speed of compiled C/C++ code, the difference has been greatly reduced, to the point where either Fortran or C/C++ will produce sufficiently fast code.

Fortran has excellent numerical library support, but C/C++ and Java have better overall external libraries. GUI building, file and network I/O, and device interfaces are more easily done with C/C++ and Java libraries. Since C and Java can use Fortran libraries with a bit of glue to massage function calls and data structures, the language choice may be forced by the necessity of dealing with data acquisition hardware, or the need for a mouse interface to the compute engine.

Fortran is difficult to use for bit manipulation, or to swap bytes to handle endianness issues. In these cases, C and Java are much better.

C++ and Java are both explicitly object-oriented, whereas Fortran has had OOP extensions added to the newest implementations. Given the nature of most scientific programming, OOP is unnecessary clutter, and should not be the basis for language choice.

Whatever language the scientist is most familiar with is probably the best choice for most numeric codes. For codes which must interface to an existing library, the language choice may be made to ease interface issues. So long as the program works, the language is basically personal preference, and nothing more.

4.2.2 Choice of Compiler

After the compiled language is chosen, probably as a matter of personal preference, the compiler needs to be chosen. Compiler choice makes a huge difference in the final speed of the binary. For modern compilers and languages, the compiler choice makes more difference than the choice of language.

For Intel-compatible processors, the Intel compilers (Fortran and C++) produce the fastest code. These compilers can use all the tricks of the CPU models, and generate significantly faster code than the more generic compilers.

For SPARC machines, the Sun compilers produce good code, as does the DEC (now Compaq) compiler. Alpha processors get the best performance from the DEC compilers.

The GNU compilers (gcc, g77, gcj, etc.) produce workable, but not blazing, code. The benefit is that the GNU compilers can produce decently fast code for nearly any processor in existence.

Intel makes its' compilers available for free to educational users, but the produced code may not be used in a commercial product. The GNU compilers are free for any purpose; modifications to the compiler source (which is also freely available under the GPL) are subject to the GPL.

4.2.3 Break a Problem into Pieces

Having decided which language and compiler to use, the task is now to implement the chosen/developed algorithm in code. Since compiled languages are so much more work than interpreted, it makes sense to implement only the bare minimum in a compiled language.

Most scientific problems that call for a compiled language do so only for a small part of the problem. Most scientific problems revolve around getting data into a program, performing a number of computations, and spitting the results out to disk. The computations are the only part that need the speed of a compiled language. All the I/O can be done at the speed of interpreted languages. Hence, the ideal solution is to implement the interface and file I/O portions of the code in an interpreted language (e.g. Java, Python, Matlab), and then call a small piece of compiled code to perform the massive computations on the loaded data. Note that this is essentially the basic design of numerical analysis environments (Matlab, Mathematica, etc.), which use an interpreter to interface with the user, and then call optimized, compiled codes to perform the math operations.

All interpreted languages of note (Perl, Python, Java, Matlab, etc.) have relatively easy methods of calling compiled code; most can call compiled code directly, as if it were a function in the interpreted language. This typically requires a wrapper function that decodes the interpreted language's data structures into the compiled language's structures, calls the compiled functions, and then encodes the results into the interpreted language's return structures. The result is easily portable and maintainable I/O code, and a minimum of compiled code.

Given the complexity and frustration in debugging large compiled programs, the overhead of writing and debugging the wrappers is more than paid for by the savings on not having to deal with the interface and I/O in the compiled language.

Compiled code is often difficult to port to new architectures (compared to interpreted languages), and hence minimizing the amount of compiled code minimizes porting issues. This is important, as scientists routinely use ancient code on new machines; the USGS has code originally written for PDP-11 machines running on modern Intel and Sun workstations. The code has been ported upwards of 4 times in its' history. This, despite the claim of the authors that the code should really have been rewritten long ago, and better algorithms used. Once a program is built and tested, it is hard to kill.

5 Scripting for Fun and Profit

5.1 Automation is Good

Scripting, which is the process of writing small programs in a *scripting language* to automate some set of commands, is a powerful technique that has been long used on UNIX. Scripting is an extension of the original concept of batch processing. The first computers were not interactive, and an operator had to set out all the commands to be performed in a command file that was fed to the computer. This file was called the *batch file*, and it allowed a computer to be used to full capacity all the time. When the computer was busy with one job, future jobs waited their turn in a queue until the computer was free. When a job was finished, the person submitting the job got their output back, after which they could then submit another job.

Scripting is the modern version of this scheme for interactive machines. Scripts collect a series of user commands into an automated format, so that running the script runs all the user commands in order. Added to this are the elements of a full programming language (scripting languages are Turing-complete), which makes scripting languages the most useful scientific tool of UNIX.

A script is ideally suited to tying multiple standard and special commands together into a single data processing algorithm. Once built, a script is a self-documenting algorithm for turning the data file inputs into interpretation outputs. In the ideal case, the script operates (non-destructively) on the original raw data, formatting, quality-checking, and processing the data into the final results that are interpreted by the scientist (including press-ready plots!). More often, scripts stop after generating final results (no plotting), and may start from partially processed data (raw data that has been hand-checked by the scientist).

Once built, a script, in addition to providing documentation on the processing steps, also makes it easy to insure that all results are of the same *vintage* (all the results come from the same processing runs, using the latest/best parameters, etc.). Plots and interpretations using differing vintages of results can be extremely dangerous and misleading.

Because a script is an automated processing scheme, it can enforce a directory structure on the data, intermediate steps, and final results. This helps keep results organized, and aids in reconstructing what happened in a project long after the project is finished.

5.2 Choice of Scripting Language

Many languages can be used to build processing scripts; user shells (bash, ksh, zsh, tcsh), Perl, Python, and Tcl are all examples of usable scripting languages. For scientific scripting, the obvious choices are user shells (typically bash or tcsh), Perl, and Python. Perl and Python are more powerful programming languages than the user shells, but their symantics make spawning external processes slightly harder, and hence they are more suited to constructing special processing tools rather than driver scripts.

For the purposes of scientific data processing, the user shells are a better choice for building driver scripts. Data processing normally ties together multiple standard and specialized tools to produce a specialized processing pipeline. The user shell semantics for building pipelines is a little easier than in Perl or Python. The user shells are also better designed to spawn external programs to perform some step. The user shells are also more commonly available on UNIX platforms.

Of the common user shells, `bash` and `ksh` are generally considered the easiest to program, although `zsh` is also a good choice. `tcsh` and `csh` can be used, but the syntax is less transparent, and the shells are less common. `bash` is installed by default on all Linux distributions, and is readily available in source and compiled form for all UNIXs. Most system administration scripts on Linux are written in `bash`, and hence most administrators know bash over other shells.

Perl, Python, Tcl, and other scripting languages could all be used to build processing scripts. These languages suffer from either more complex syntax, less common availability, or both. Still, they will all solve the same problems, with similar or identical performance.

6 Geology & Geophysics High-Performance Computing

For some large computing jobs, a single computer, regardless of language, is too slow. These problems can sometimes be solved faster on a cluster of machines each of which solves only part of the problem. This sort of approach, known as *parallel computing*, is worth the price of implementation only for the very largest problems.

In Geology & Geophysics, we have access to two parallel computing clusters: `terraflop` and `icebox`. Terraflop is run by the department, and consists of a handful of (slow) nodes running Linux. Icebox is a large cluster, administered by the Center for High-Performance Computing (CHPC) in INSCC. Icebox is very large (>200 nodes), with massive disk and CPU available. Use may or may not be free, depending on the resources required and the loda on the cluster at the time.

For information and help on using terraflop, see the document available at `http://thermal.gg.utah.edu/facilities/cluster`. Information on the Icebox is available from CHPC.