arXiv:0905.1628v2 [physics.comp-ph] 15 Jun 2010

Lecture given at the International Summer School Modern Computational Science
(August 9-20, 2010, Oldenburg, Germany)

Scientific Software Engineering in a Nutshell

Helmut G. Katzgraber

Department of Physics and Astronomy, Texas A&M University
College Station, Texas 77843-4242 USA

Theoretische Physik, ETH Zurich
CH-8093 Zurich, Switzerland

Abstract. Writing complex computer programs to study scientific problems requires careful planning and an in-depth knowledge of programming languages and tools. In this lecture the importance of using the right tool for the right problem is emphasized. Common tools to organize computer programs, as well as to debug and improve them are discussed, followed by simple data reduction strategies and visualization tools. Furthermore, some useful scientific libraries such as boost, GSL, LEDA and numerical recipes are outlined.

Contents

1 Introduction

Taking on scientific problems using computers, while not completely new, is still one of the branches of scientific research which is least well documented—especially for beginners. There is no "right" or "optimal" way to develop software to study scientific problems, because this is an extremely problem-dependent task. For example, studying events in high-energy physics experiments requires thousands of independently-working CPUs, simulations of hydrodynamic processes in general require massively-parallel machines, whereas many of every day's numerical problems we encounter can be solved on a simple desktop workstation or laptop computer.

The goal of this tutorial is two-fold: First and foremost, to convey the necessary tools and "organizational skills" to develop (small-scale) computer programs for scientific simulations. The tutorial focuses on different aspects one should pay attention to before, during, and after the software development phase. Second, emphasis is placed on the use of the right language for the right problem. For example, it makes little sense to write a program in a programming language such as C or FORTRAN to numerically integrate a function, if software packages such as Mathematica or Matlab can do this within one line of code.

Clearly, it is impossible to convey all the necessary information in this brief tutorial. Thus emphasis is placed on the necessary references where students can search for and find the necessary information. In this tutorial it is assumed that the reader is familiar with a basic *nix environment (e.g., Linux, HP-UX, MacOS X, . . .) and has knowledge of at least one high-level programming language (e.g., FORTRAN, C, C++, Java, . . .). Unfortunately, it is beyond the scope of this tutorial to introduce the *nix environment or a high-level language. The reader is thus referred to a vast variety of freely-available online tutorials, as well as the book "*Learning the Unix Operating System*" by Peek *et al.* in the O'Reilly Bookshelf [1]. If necessary, examples are written in the C programming language and shell commands are presented using bash shell syntax [2]. General notation: Source code and commands are typeset `in this font` and a *nix prompt in a shell is represented by "`>`."

2 General strategy

In general, writing a computer program to study some scientific problem can be divided into different phases: planning, coding, debugging and testing, data production, data analysis, documentation for posterity and publication of the results. First, the general strategy for software development is discussed, followed by some useful tools.

General rule: Never simply start writing a program for some research project. Most of the time, programs written in an ad-hoc way turn out to be "Franken-programs" that are hard to read, difficult to adjust to different problems, and even slow. This is particularly the case when more than one researcher is working on a given software project. Therefore some *software engineering* should be done. Below, the basic steps of software engineering within a scientific context [25,33] are outlined. See, for example, Ref. [25] for further details.

2.1 Definition and outline of the problem

Probably the first task to accomplish is to decide which quantities need to be computed using which algorithm/method to study a given computational problem. Think also about other quantities which *could* be measured (at small computational cost) that might be of use for later projects and might not be necessarily relevant for the current problem. Some points to consider:

☐ Draw a flowchart (diagram) for the problem. Think about the input, routines needed (and their order), as well as the output. Make sure you are using the right algorithm and programming language to solve the problem (check the literature and talk to your peers!).

☐ Make a list of necessary input parameters for the simulation. Parameters that change often or alter the functionality of the code can be passed as run-time options (e.g., space dimension, seed of the random number generator, data compression option, etc.). It is recommended to use a parameter file for all other parameters. Keeping such a parameter file together with the produced data is of paramount importance to ensure data provenance (discussed later).

☐ Because the cost of storage has decreased considerably in the last few years, it is recommended to store as much simulation information as possible. Thus, make a list of potential observables (quantities to be measured), and, as mentioned before, any other observables that might be useful elsewhere, and decide to what level data have to be stored. For example, in a Monte Carlo simulation one can store the final thermal average of some quantity, one can store the time evolution (logarithmically-spaced) of the same quantity, or a configuration snapshot of every single step in the simulation. Clearly, the storage requirements increase considerably.

- ☐ Identify *objects* and *data structures* in your problem. This shall enhance the modularity of the program. For example, position and velocity of a particle can be combined into one structure dubbed "particle" thus allowing for an easy change of coordinate systems or space dimensions. Use a bottom-up approach to programming: Define essential routines and code them up first. Add the "skeleton" of the simulation at the end.
- ☐ Think back: Do you currently have programs or libraries that would be of use for this project and can be included? Think ahead: Will this program be used in the future for other projects? If so, ensure that extensions can be trivially accomplished and do not require a complete re-writing of the software.

Once the planning phase is complete, it is important to decide which language to use for a given project.

2.2 Selecting a programming language

In general, one can roughly classify computer languages for scientific applications into three (overlapping) categories: high-level programming languages (such as C, C++, FORTRAN, Java, etc.), scripting languages (e.g., Perl, Python, shell, R) and "symbolic languages" (e.g., Mathematica, Matlab, Maple, to name a few). It is of paramount importance to select the right language for a given problem. Keep the following points in mind when selecting a programming language:

- ☐ What is the numerical effort in CPU hours you expect the project to take? If this number is very large, then a fast high-level language should be used unless the complexity of the problem requires the use of a symbolic language. Furthermore, large projects can be sped up by using parallel programming techniques on large clusters. In general, these Message Passing (MPI) libraries [3] only exist for high-level languages and some software packages such as Matlab.
- ☐ Does the problem require a real-time graphical display of data or cross-platform compatibility? In that case Java (by Sun Microsystems) might be the best choice. Note that Java is a (relatively slow) object-oriented language.
- ☐ Does the problem require parsing and analyzing large data sets? In this case a scripting language such as Perl or Python might be most useful because of the built-in Regular Expression parsing capabilities [4]. Furthermore, in the case of Perl, input-output tasks are trivially accomplished and so the combination of different data sets or files for parsing is done very efficiently. Note that these languages do not require compilation and thus are very portable between operating systems and different hardware architectures.

- ☐ Does the problem require symbolic manipulations of functions or, e.g., complex numerical integrations of functions? In this case a symbolic language such as included in Matlab or Mathematica might be the optimal choice to tackle the problem. For example, Mathematica allows for axiomatic definitions of operators. This in turn simplifies considerably the computation of, e.g., Feynman diagrams for large-order expansions.

- ☐ Does the problem use pre-defined software packages that require being called with specific parameter combinations? In this case it might be most useful to "wrap" the used software in a shell (or Perl) script to automatize the (often tedious) tasks involved in calling the program. Furthermore, multiple analysis steps can be wrapped in one script thus reducing effort and ensuring a minimal error rate.

Overall one has to consider the *total* amount of time spent on a project, including the software development. For example, why write a C program to integrate a function if Mathematica has a (very good) built-in integration routine? This could save considerable time in the solution process of a problem.

A typical example of how to split up tasks between programming languages is illustrated with a Monte Carlo simulation of a spin system. A good approach would be to write the main routine generating the data using C or C++. To run the program written in a high-level language on many workstations, one would use a shell script to farm out and monitor the executables. To analyze the data—such as for example computing thermal averages over measurements and error bars—one could use Perl scripts for post-processing.

2.3 Writing the program

Using a consistent notation for variables, as well as a clear pre-defined style of programming ensures the portability and—most importantly—readability of the code after extended periods of time. General considerations:

- ☐ Use a modular programming approach: Split the code into several modules that can be stored in independent source code files. This has the following advantages: It allows you to easily use these modules for other programs. For example, a C-routine to compute the mean of an array of numbers can be stored in a file `mean.c` which then can be used for other coding projects by simply copying the file or loading a library (discussed later). If you have a large software project, changes to individual modules will only require recompiling of these object files. Finally, if multiple researchers are working on one project, having only one source file prevents people from working on the same file in parallel.

- ☐ To keep the program logically structured, separate generic data structures and algorithms in separate files, such as `.c` and `.h` files in C.

- ☐ Give meaningful names to variables, routines, etc. For example, a subroutine `e(int *s)` carries little to no information for the reader. Instead, naming the routine and arguments in the following way `energy(int *spins)` is far more informative. Be sure you use a *consistent* notation. Example:

  ```
  #defines USING_ALL_CAPS
  variables in smallCamelCase
  local variables _withUnderscore
  classes in CapitalizedCamelCase
  functions small_with_underscores( )
  ```

  If you choose to use short names for variables, be sure to document their meaning.

- ☐ Try to use proper indentation when writing routines as this increases readability considerably. Good style:

  ```
  #include <stdio.h>

  int main(){

      int counter;

      for(counter = 1; counter < 100; counter++){
          printf("Asterix beats the Romans up. Ha!\n");
      }

       return(0);
  }
  ```

  Bad style:

  ```
  #include <stdio.h>
  int main()
  {int i;
  for(i = 1; i < 100; i++) printf("Asterix beats the Romans up. Ha!\n");
  return(0);}
  ```

  Adding extra space *always* makes the code easier to read, *but* do not go too crazy as then it might be too hard to read, too. A good measure is a four-character indentation as shown in the example above. Most editors do this automatically.

- ☐ Avoid logical jumps (goto) in the program. Debugging will be a nightmare otherwise.

- ☐ Be careful when using global variables. It makes sense to define some global variables, but, in general, it is *highly* recommended to pass variables via a "global-variables" structure to avoid programming mistakes. Quantities that do not change in the simulation and which are needed by many routines, could, in principle, be defined globally (e.g., the space dimension or the number of particles).

One of the most important steps in writing software is to properly document it. Always write at the beginning of the routine a short introduction about the routine and what it does. You can also add a brief history with different versions and revisions. List parameters as well as their meaning. Nice examples are shown in Ref. [25]. Within the code, list the meaning of variables, comment on what certain routines for a code block do and always remember that a few years down the road it might not be quite clear what you were doing. Furthermore, provide external documentation for other users (generally in form of a README file).

Finally, you might wonder in which environment/editor the code should be written in. While some developers rather use IDEs (integrated development environments) such as *eclipse* [5] or Apple Inc.'s *Xcode* [6], others prefer simple *nix editors such as *emacs* or the *vi* editor. This again depends strongly on the scope and complexity of the project, as well as personal preference and habits.

2.4 Help! I am lost!

Do not despair. Almost all programs have built-in help. In general, when these programs are started from a command line, you can obtain a bare-bones help by using the following *flags*:

```
> program -h
```

or

```
> program --help
```

Always try both since sometimes one is more verbose than the other. In addition to the built-in help, any *nix operating system has a built-in system of manual pages. The system can be started in a shell by issuing the command `man`. For example, to find out more about the `man` command itself, simply issue:

```
> man man
```

Scrolling is accomplished with the arrow keys, searching within the man-page for a text string "`string`"can be done with "`/string`" and quitting the system is done by pressing "q". If you are not quite sure what you are searching for, you can issue the wild card search command, here for the string `cron`

```
> man -k cron
```

which produces:

```
cron(8)                 - daemon to execute scheduled commands
crontab(1)              - maintain crontab files for users
crontab(5)              - tables for driving cron
```

The output has a small description. Note that there are two man-pages for "crontab." To access page 5, simply issue

```
> man crontab -S5
```

Certain core components of the standard C library are also documented via man-pages. Therefore, when programming in C, information about any built-in function can be obtained. "`man sin`" returns information on how to call the sine function, which arguments the function call has, as well as which header files to load (in this case `math.h`).

Finally, there is always the internet and Wikipedia as sources of information. Be always aware of what the sources are since there is no guarantee that the information retrieved is correct.

3 Provenance

In the previous section the importance of readability of the code as well as documenting the code has been emphasized. One of the most important reasons for careful programming is code and data *provenance*. In the Oxford dictionary *provenance* is defined as "a record of ownership of a work of art or an antique, used as a guide to authenticity or quality." While this definition is explicitly given for antiques and pieces of art, the concept of provenance has become of increasing importance in the last few years within the context of computational sciences. As computer programs become increasingly complex and data sets grow exponentially with time, it is of paramount importance to document the *origins* of the code and data, the different *revisions* and versions, as well as the *usage* of code and data analysis tools.

Although there are no world-wide standards on how to properly store computer codes or data files—each of which depend highly on the application or problem—, there have been some incentives to standardize at least data files to ensure that results published in scientific journals can be reconstructed in the future. In this section we differentiate slightly between source code provenance and data provenance. Probably the easiest way to ensure source code provenance is to use a version control system which keeps track of changes made to software.

3.1 Version control with subversion

In addition to source code provenance, version control systems (VCSs) have the advantage of allowing one to catalog in an easy way older versions of a given document (text file, program, etc.) for retrieval. This is particularly useful when more than one person is working on a program since then individual portions of the code can be *checked out*, ensuring that two persons are not editing the same file at the same time. Once editing is complete, the code is *committed* to the VCS for others to access.

Subversion [7] is a VCS that can manage any project situated on a directory containing any arbitrary files on your hard disk. Note that the system does not store copies of all revisions made, it merely stores the *changes* between the different versions. By default, subversion does not *lock* the files, i.e., more than one user can edit a given file. If this is undesired, file locking can be turned on.

To create a repository located in the directory "`/home/user/repository`" issue the command

```
> svnadmin create /home/user/repository
```

All operations performed on the repository are done with the command `svn`. The general syntax is `svn <subcommand> [options] [args]`. More help can be obtained by simply typing `svn help`. If you already have some files in a directory called "`lala`," you can *import* them to the repository by specifying a destination directory as well as a *message* via the `-m` option:

```
> svn import lala/ file:///home/user/repository/lala -m "first import"
```

Note that we use a universal resource locator (URL) syntax for the placement of the file, in this case "`file://`" since it resides on the local disk. This means that networked projects can be accessed via "`http://`" or "`ftp://`" (check the documentation). To view the files in the repository and within the created directory "`lala`" type:

```
> svn list file:///home/user/repository/

> svn list file:///home/user/repository/lala
```

Before you start editing any files be sure to check out the files with

```
> svn checkout file:///home/user/repository/lala
```

or any specific file within the repository by specifically specifying it. You can check the status of which files have been edited by using the command

```
> svn status
```

Once you have completed the necessary edits of the files, you can *commit* your changes and upload them to the repository via

```
> svn commit -m "did some lousy changes"
```

Note that again a message is passed, documenting the changes made. *While* you have a project checked out you can always see the changes you have made with

```
> svn diff
```

The output is in the Unix diff format which allows you to patch a file with the provided text (see `man diff` and `man patch`). If other authors performed changes to the repository's files, you can obtain the latest revision by using the command "`svn update`." Potential conflicts can be resolved using the "`resolve`" option. Finally, the revision history can be generated using "`svn log`." There are many other options and the reader is referred to the documentation as well as Ref. [23].

Finally, there are also distributed source control management tools. For example, Mercurial [8] ensures that each developer has a local copy of the source code and the entire development history. While subversion relies on a central server to store the revisions, code development on Mercurial is independent of a central server and therefore also network independent. Note that Mercurial is written in Python and therefore platform independent, as well as open source.

3.2 Data provenance

In addition to the need for proper documentation of source codes, it is imperative to have consistent data formats for numerical simulations that are well documented with external READMEs. Currently, it is unclear what the best format for data storage is. Large and complex data sets are generally binary compressed and stored using *hdf5* [9]. For now, we assume that this is not the case and that storing data in plaintext files is appropriate. The optimal approach to storing data is using XML tags (see books on the extended markup language for details, e.g., Ref. [28]) with the disadvantage that data files are extremely large because the XML tags require a nonnegligible number of bytes. In addition, data stored in XML format are generally hard to read and require parsers to produce sets which are easily understood. Therefore, a good compromise solution is to generate a well-documented tabular data structure. A typical bad example on how data are stored in tabular manner is given below:

```
2.0000000000e+00 -1.7430877686e+00
2.2000000000e+00 -1.5684322497e+00
2.4000000000e+00 -1.3377984675e+00
2.6000000000e+00 -1.1194154460e+00
```

There is no description of the data, there are no tags of any kind, there is no information on how or when it has been generated. Therefore, it is impossible for a third-party to understand the results. In this case, it is the temperature-dependent internal energy for a two-dimensional Ising [26, 35] model with 8^2 spins computed using Monte Carlo methods. There is no way to check in hindsight if the data are converged since only averages at a given number of Monte Carlo steps have been stored. To be able to ensure that the data are converged, a whole new simulation would have to be performed.

By adding a (parsable) header file to the data set, it is easy to know what is stored in the file. This is shown in the example below (excerpt only) which has such a header with the simulation parameters, as well as the process ID in case one needs to kill a specific simulation. Furthermore, the temperatures are listed (for parsing and analysis convenience) as well as the average energy data as a function of (logarithmic) Monte Carlo time to be able to check if the data are converged. Time-dependent blocks for different temperatures are separated by empty lines.

```
# 2D ferromagnetic Ising code. h. katzgraber 12/06/2009 (v4.20)
# simulated using simple Monte Carlo
#
# system size (number of spins)         L  = 8 ( N = 64 )
# space dimension                       D  = 2
# max exponents for sweeps (EXP_MAX)       = 12
# maximum equilibration/measurement time   = 4096
# initial seed for this run                = 70893
# process id (PID) for this run            = 89993
# temperature set:
#
# | 2.0000
# | 2.2000
# | 2.4000
# | 2.6000
#
# T    MCS     <energy>
2.0000 2      -1.5312500000e+00
2.0000 4      -1.8281250000e+00
2.0000 8      -1.7812500000e+00
2.0000 16     -1.7734375000e+00
2.0000 32     -1.7851562500e+00
2.0000 64     -1.7900390625e+00
2.0000 128    -1.7685546875e+00
2.0000 256    -1.7634277344e+00
2.0000 512    -1.7650146484e+00
2.0000 1024   -1.7451782227e+00
2.0000 2048   -1.7450561523e+00
2.0000 4096   -1.7430877686e+00

2.200 2 ...
```

Note that the data file also includes a version of the used program. If the source code used for the project is under continuous development, it is very important to know which version of the program was used to generate a given data set. Clearly, there is no best recipe since the way data are stored depends highly on the problem studied. Nevertheless it is crucial to put some thought into how much of the data and how the data are stored for posterity and provenance. Furthermore, always document the data, software and analysis tools.

4 Compiling, debugging, profiling, testing

Once the code is written, it will likely not compile. That is the best case scenario since compilers, in general, deliver useful error messages. If the code compiles, and crashes during the first run, this is not too bad either since debuggers can assist in finding the problem. If the code compiles and the program exits normally, but the data you produce do not seem to be correct, be very afraid since these are the hardest bugs to find. In this section compilers, options, and basic compilation procedures are discussed, followed by common debugging tools. In addition, some suggested approaches for finding hidden problems *if* the code compiles and runs are presented.

4.1 Compilers and options

There are many different compilers for many different languages. For this reason in this section we focus on the GNU C compiler `gcc`. There are commercial compilers (Intel, Portland Group, Numerical Algorithms Group) that, in general, produce considerably faster executables than the GNU C compiler. Hence, if you have access to these (e.g., on a large computer cluster) you should try to use them.

After the code is written, it is compiled with a compiler. The compilation process encompasses several stages, starting with the preprocessor which resolves `#define`, `#include` and `#if` directives. In this case `gcc` actually invokes the preprocessor `cpp` to do the preprocessing. Once the preprocessing is completed, the compiler produces machine-readable assembly language from the input files (usually this step produces no visible output, but, if requested, it will produce assembler files with a "`.s`" ending). Within `gcc` the assembler `as` takes the produced *assembler* code and generates object files (with the ending "`.o`"). In the final stage, the objects `.o` are placed in their proper place of the executable. Library functions might be included at this stage as well. `gcc` invokes internally the *linker* `ld` for this task. Note that, in general, all these steps are performed automatically by the compiler. You *do* have the option to stop the compiler at any stage and invoke the individual steps manually. Because this would be beyond the scope of this lecture, the reader is referred to Ref. [27].

Basic compiler use The most basic way to call the compiler is in the following way:

```
> gcc main.c lala.c momo.c
```

This produces an executable `a.out` which then can be executed. Programmers rarely build everything at once. Usually they attempt to compile individual parts of the program to check for errors before the whole program is compiled. Thus, conversely, we can call

```
> gcc -c main.c
> gcc -c lala.c
> gcc -c momo.c
> gcc main.o lala.o momo.o
```

to obtain the same result. The "`-c`" options means "compile but do not link." At the end `gcc` is invoked with only object files to henceforth link them and produce an executable.

If library functions are to be linked into the executable, we need to call these with the "`-l`" option. In this case, if the library is called for example `libmy.a` and it is needed for the object compilation of `main.o` we need to issue

```
> gcc -c main.c -lmy
```

Libraries are discussed in more detail later. If we want to include an include file or library which is in a nonstandard directory, we need to tell the compiler where to find these. For example:

```
> gcc -c main.c -I/home/user/myinclude -L/home/user/mylibs -lmy
```

Summary of some compiler options There is a whole zoo of compiler options that influence the behavior and speed of your executable, as well as show in a verbose manner warnings and comments during the compilation of the program. There is a core subset of compiler options which not only are very important, but also more or less carry over to different programming languages and compilers. In what follows the most important compiler options for the GNU C compiler (`gcc`) are listed. For a complete list, `man gcc` as well as Ref. [27] can be used.

`-v`
: Prints the compiler version and details about the configuration.

`-o`
: Sets the name of the output file.

`-c`
: Compile only, do not link. Produces object files.

`-lany`
: Tells the compiler to link the objects to a library called `libany.a`.

`-static`
: Link only to static libraries. This is sometimes necessary for executable portability to other machines where the libraries are not installed.

`-Wall`
: Turn on (almost) all compiler warnings. The compiler warns if there are potential problems in the code.

`-g`
: Turn debugging on and generate an expanded symbol table. Only code compiled with this flag produces meaningful output in common debugging programs such as `gdb` (see Sec. 4.3). The symbol table can, in principle, later be removed with the Unix `strip` command.

`-pg`
: Link the program for profiling with `gprof`. This produces execution statistics and timing information.

`-D`
: Defines macros (see compiler documentation). Useful for debugging or making architecture-dependent code.

`-L` and `-I`
: Tells the compiler where to find custom library and header files (explained above), respectively.

`-O`n
: Optimizes the code to level n. $n = 0$ means no optimization, whereas $n = 1$ means that the compiler tries to reduce the size of the code as well as execution time. Compilation is slower and requires more memory. For production, it is optimal to compile with $n = 2$, i.e., `-O2`. In this case the code is considerably faster than when no optimization is used. $n = 3$ should be thoroughly tested before used for production as "experimental" optimization features are turned on. If possible, avoid. There are further custom optimization options such as `-ffast-math`, `-finline-functions`, `-funroll-loops`, . . . , as well as architecture-specific instructions. Details can be found in the compiler documentation.

In general, when it comes to speeding up an executable, a careful choice of compiler options can yield great speed improvements. Loop unrolling and inlining, combined with `-O2` give the greatest performance boosts.

4.2 Make

The `make` facility [31] is one of Unix's most useful tools. Unfortunately, it is one of the least used by beginning programmers. Putting it in simple terms, `make` is a programming language to automatize large compilations. Not only does `make` handle automatic compilation of computer programs, the compilation of large LaTeX documents whilst including tables of contents as well as bibliographies can be simplified enormously by using `make`. `make` checks if certain files have changed and, after checking pre-defined dependencies in a configuration file called *Makefile*, will *only* compile the necessary files and perform the final linking. Not only does this considerably speed up compilation time for large projects, it also reduces possible errors when linking to old object files.

The *Makefile* contains the information about the different *dependencies* between the source files as well as the necessary *commands* to deal with these. In general, the goal is to complete a task called a *target*. A pair of dependencies and commands is referred to as a *rule*. General syntax:

```
target: sources
<TAB>   commands
```

The first line contains the dependencies for a given target, the second one the commands to perform. Note that the command line *must* always begin with a tabulator shift (`<TAB>`). In all following examples this `<TAB>` is not shown explicitly. Example:

```
all: main.o lala.o
        gcc -o runme main.o lala.o

main.o: main.c program.h
        gcc -c main.c

lala.o: lala.c
        gcc -c lala.c
```

In the previous example the file `main.o` has to be compiled if either `main.c` or `program.h` have changed. On the command line you run `make` via

```
> make
```

If you want the program to be built, you would do `make all`, and if you just want to recompile `lala.c`, you would do `make lala.o` Whenever the date of the source files is newer than the date of the targets, `make` will recompile these. In the following case (which is only practical for small projects), all files are compiled every time:

```
main:
        gcc -o runme main.c lala.c
```

`make` checks recursively the dependencies in the Makefile and executes the necessary commands. `make` also allows variable definitions (*macros*). For example, one could define

```
DEBUG = -g -Wall -ansi
SRC   = *.c
```

Then, these macros could be used later in the Makefile allowing for easy global replacements of, e.g., the compiler or debugging flags:

```
all:
        gcc -o runme $(DEBUG) $(SRC)
```

The behavior of `make` can also be altered with run-time flags. Here is a list of the most convenient:

`-f` *filename*
Normally, `make` looks for a file called *Makefile*. With this flag the name can be changed to any *filename*.

`-n`
Do not execute any command, simply list what would be executed (debugging).

`-i`
: Normally, `make` terminates if it encounters an error. This options forces `make` to continue with the other remaining tasks.

`-j` *n*
: Run *n* commands at once (useful on multiprocessor machines).

There are far more options and complexity to the `make` command (see the manpage and Ref. [31]). Note also that line carriages can be accomplished with a "\".

The one-size-fits-(almost)-all Makefile For most applications, the following Makefile—while requiring recompilation of the whole source—provides the basic functionality and efficiency needed. Keep in mind: commands start with a `<TAB>`:

```
# check the hostname (hashed lines are comments)
HN := $(shell /bin/hostname -s)

# set executable name
EXECNAME    = runme

# default compiler settings
CC          =  gcc
OPT         = -O2
DEBUG       = -g -Wall -ansi
LDFLAGS     = -lm
INCLUDE     = -I/usr/include -I../include

# on moo.tamu.edu we use the intel compiler
ifeq ($(HN),moo)
    CC      = icc
    OPT     = -O2 -static
endif

SRC         = *.c
OBJS        = $*(SRC).o

# generic compilation for data production
$(EXECNAME):
	$(CC) $(OPT) $(SRC) $(INCLUDE) -o $(EXECNAME) $(LDFLAGS)
	/bin/rm -rf *.o

# debugging compilation
db:
	$(CC) $(DEBUG) $(SRC) $(INCLUDE) -o $(EXECNAME).db $(LDFLAGS)
	/bin/rm -rf *.o

# clean up
clean:
	rm -rf *.o core *~ $(EXECNAME) $(EXECNAME).db
```

The aforementioned Makefile shows some crucial ingredients needed when compiling simple projects on different machines. Because some compilers are only installed on certain machines, in line 2 we check the hostname of the machine using a shell call. In line 5 we set a variable with the name of the executable and in lines 8 – 12 we define variables for the compiler, libraries, optimization, etc. From lines 15 – 18 we check for a specific hostname with an `ifeq` statement. If the code is compiled on `moo` then we change `CC` from `gcc` to `icc`. Note that we also perform a static compilation to ensure that all libraries are contained in the final executable. For simplicity, all source files are included using a wild card in line 20 and in line 21 the object files are defined by a replacement of the file ending from `.c` to `.o`. Production compilation is accomplished by calling `make` or `make runme` in lines 24 – 26; the debugging compilation is called in lines 28 – 31. Note that after compilation the object files are deleted since we want to ensure that the latest objects are included every time the code is compiled. In lines 24 – 25 we add a directive to "clean up" that deletes binaries, core files and object files. Again, this is not practical if the compilation takes several minutes. But if your code compiles in 5 – 10 seconds, this is possibly the easiest and most versatile Makefile to use.

4.3 Debuggers

In this section we describe only the GNU `gdb` debugger. There are graphical front-ends (such as `ddd`) for this debugger, but these are not discussed here for the sake of brevity. `gdb` lets you run a C or C++ program, stop execution within the program, examine and change variables during execution, call functions, and trace how the program executes. Only the basic operation is presented, more information can be found online at the Free Software Foundation's *Debugging with GDB*, in Ref. [27], the manpage, as well as within the program by issuing the command "`help`."

For a program to be debuggable, one needs to use the "`-g`" compilation flag. Note that this makes the program very slow, i.e., do not use this flag for production runs. Suppose a program `runme.db` has been compiled. The debugger is invoked via

```
> gdb ./runme.db
```

The source code can be listed within the debugger with the `list` command. To run the program within the debugger, simply issue the `run` command. If your program requires arguments, simply list them: `run -p params.in`. To have the execution stop at a specific line, one can use the `break` command, i.e., `break 42` stops execution at line 42 of the source code (note that the break-point needs to be set before the command is run). You can `print` the contents of a variable or array, for example

```
> (gdb) print array[10]
  $2 = 72
```

whenever execution is stopped. Similarly, you can also `set` the values of variables or arrays:

```
> (gdb) set array[10] = 43
```

To continue the execution of the program step-by-step, use the `next` command; if you want the program to simply execute normally (until it encounters the next breakpoint) use the `cont` command. Note that `next` executes an entire function if it encounters a call, while `step` enters the function and keeps going one statement at a time. This allows the programmer to fully control each step of the program to ensure that things are running according to plan.

Because the breakpoints are numbered, during the execution conditions can be placed with the `condition` command. It allows one to stop execution if a condition is met – something which is very useful when testing. Individual functions in the program can be called with the `call` command and `info` delivers *any* information about breakpoints, registers, variables, etc. Finally, all aforementioned commands can be abbreviated by simply typing the first letter of the command only. For example `n` has the same functionality as `next`. Further details about the use of `gdb` can be found in Ref. [27] and in the documentation.

4.4 Memory debugging with valgrind

If memory allocation is not done carefully, memory segments outside a data structure could be addressed. In this case the program exits abnormally with a `Segmentation fault` dumping a *core*. In addition, forgetting to free memory segments (also known as *memory leaks*) could quickly use up all the available RAM on a computer and crash it. Because memory allocation errors often do not cause the program to crash immediately at the point where the error happened, these types of errors are difficult to find. The reason for this seemingly erratic behavior is that we cannot directly influence memory management of the operating system. In some cases, writing data out of bounds might overwrite other variables causing a crash of the program, in others not. This is the reason why you should *always* test a program with memory checkers even though things might seem to be running correctly.

To successfully trace memory errors, memory checkers such as valgrind [10] are very useful. Valgrind is freely available online. To debug a program `runme.db` simply type on the command line

```
> valgrind ./runme.db
```

Again, to be able to determine where in the source code the bug is located, it is important to compile with the `-g` flag. If you use the "`--db-attach=yes`" flag, valgrind starts the debugger for you to analyze the program in more depth. It is recommended to use valgrind in verbose mode (use the `-v` flag) to obtain detailed information about the different memory problems the program finds. To see a sample output of valgrind, run the memory checker on the *nix program of your choice, e.g., `valgrind date`:

```
> valgrind date
==6147== Memcheck, a memory error detector.
... further copyright output
==6147==
Tue Apr 14 20:51:27 CEST 2009
==6147==
==6147== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 15 from 1)
==6147== malloc/free: in use at exit: 0 bytes in 0 blocks.
==6147== malloc/free: 9 allocs, 9 frees, 1,102 bytes allocated.
==6147== For counts of detected errors, rerun with: -v
==6147== All heap blocks were freed -- no leaks are possible.
```

The number at the beginning of each line is the process ID and can be ignored. For the particular case of `date` no errors were detected. If errors are present, the output mentions mainly the error count:

```
> valgrind ./runme.db
==6745== Memcheck, a memory error detector.
... further copyright output
==6745==
==6745== ERROR SUMMARY: 27 errors from 2 contexts (suppressed: 0 from 0)
==6745== malloc/free: in use at exit: 0 bytes in 0 blocks.
==6745== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==6745== For counts of detected errors, rerun with: -v
==6745== All heap blocks were freed -- no leaks are possible.
```

Re-running valgrind with the `-v` option (considerable output) shows exactly where the problems are located. More information on valgrind's options can be obtained by using `valgrind -h`.

4.5 Testing, testing, testing, ...

Once the code compiles and runs, it still does not mean that it is producing the correct results. At this stage, if there is suspicion of a problem, different checking mechanisms can be used.

In C programs, one can use `#if` directives to have the preprocessor include certain code segments. This can be very useful for debugging purposes. For example, in a program

```
#if (DEBUG == 1)
    printf("DEBUG: energy = %f\n",energy[i]);
    assert(energy[i] < 0);
#endif
```

The behavior of the debugging code can then be controlled by setting the necessary variable in the header of the program:

```
#define DEBUG 1
```

Whenever `DEBUG` is `1` the debugging information is compiled into the code and executed, otherwise not. This allows to track the behavior of certain variables where potential values are known from scientific insights. For example, in the C code above, if the energy is positive [the test is performed with the built-in `assert( )` C function; for C++ use `cassert( )`] the program aborts abnormally.

Once the program seems to produce reasonable results, one should *always* perform *science consistency checks* and, if possible, compare against known data. Note that the latter is not always possible. Let us assume we are simulating a two-dimensional ferromagnetic Ising model [35]

$$\mathcal{H} = \sum_{\langle ij \rangle} J_{ij} S_i S_j , \qquad S_i \in \{\pm 1\},$$

on a square lattice with nearest-neighbor interactions $J_{ij} = -1$, using Monte Carlo methods. A simple back-of-the-envelope calculation show that the *energy per spin* should converge to -2 for temperature $T \to 0$. This follows from the fact that the energy is minimized when all the spins point in the same direction. In that case, $E = -2N$, where N is the number of spins. It is straightforward to simulate the model at close-to-zero temperature for a tiny system to verify if the result for the energy per spin is -2. If this is the case, it is likely that there is still a bug hidden somewhere in the program, but the core routines are likely bug free. Another option is to reduce the program to study a problem with known results. For example, if a two-dimensional spin-glass is being studied, changing the random interactions between the spins ($J_{ij} \in \{\pm 1\}$) to a ferromagnetic interaction ($J_{ij} = -1$) allows for a comparison with known results for the two-dimensional Ising model.

4.6 Program timing and profiling

Once the program is bug-free, it is time to ensure that it runs as fast as possible. For example, the simulation of spin glasses at finite temperatures [22, 24, 30] usually takes a large amount of time. A typical project requires 10^5 or more CPU hours that corresponds to approximately 12 years. If we can ensure that our program runs 10 times faster, this would mean that execution only takes 1.2 years. Therefore *profiling* programs to find parts which are executing slowly is of great importance.

time The simplest way to time a computer program is to use the *nix built-in `time` command:

```
> time ./runme
```

The syntax of the output depends on the *nix flavor used. In all cases the following three numbers are returned: `real` time, which is the time it took the process to complete, `user` time, which is the time the process spent in user space, and `system` time, which is the time the process spent performing system calls.

One can also use the C function `time( )` in the program to perform one's own timing. This is done by including "`sys/times.h`" in the header of the program. While this results in total running times of the program and, if properly implemented, time step measurements, it does not return information on how much time was spent on each individual subroutine.

gprof The GNU graph profiler `gprof` [27] provides a more detailed analysis of the program. For a program to be profiled, it needs to be compiled with the "`-pg`" flag:

```
> gcc -pg -o runme.prof main.c lala.c
```

Once the program has been compiled, it can be run in the usual way:

```
> ./runme.prof
```

In addition to the usual output, a file `gmon.out` is produced. This file contains the necessary profiling information. At this point, the profiler is invoked with

```
> gprof runme.prof gmon.out >& output.txt
```

Note that the output is delivered to the standard output and contains a wealth of information, which is why we pipe it into a file `output.txt`. The most relevant information is the table at the end which shows the (flat) profile listing all functions and how much relative time was spent on them (first column):

```
%      cumulative   self              self     total
time    seconds   seconds    calls  ms/call  ms/call  name
17.7       3.72      3.72 13786208     0.00     0.00  Append [8]
 6.1       5.00      1.28   107276     0.01     0.03  MkPath [10]
 2.9       5.60      0.60  1555972     0.00     0.00  StringFree [35]
```

The second column contains the cumulative time added up from the time each routine used in the third column. The fourth column lists how many times a routine was called. The fifth column contains the average number of seconds per call to a function, i.e., the third column divided by the fourth. The final column has the total time spent on a function and its descendants. The output also contains a *call graph* that shows all functions and dependencies (not discussed here, see the man page). In general, it is easy to spot where the simulation is spending most of the execution time. If this is not in the core of the program, then there is likely a slow or cumbersome implementation of a function which needs to be optimized.

gcov GNU's `gcov` is a test coverage program [11] that describes the degree to which the source code of a program has been tested. It therefore allows you to discover which parts of your program have not been tested. While gprof tells you how much time is spent in a certain part of the code, `gcov` tells you which lines of the code are actually executed and how often. Using `gcov` is simple: Compile your program in the following way

```
> gcc -fprofile-arcs -ftest-coverage main.c
```

Execute the program (this will generate additional files needed by `gcov`) and then run `gcov` on the individual c files of your routines:

```
> gcov main.c
```

The last step generates a file `main.c.gcov` which contains the necessary information to further optimize your program. In particular, it contains the number of times a function, statement, or line of the code has been executed.

5 Running the code

Before the large production runs are started, it is important to check how many resources the project will require. For example, if for a large system the code runs longer than what is allowed on a supercomputer's queue, the program will be terminated before completion. Similarly, if the memory requirements exceed the RAM on the computer, the machine will crash. And if the output exceeds the free disk space on the computer, a crash will likely occur as well. A simple approach is to perform *scaling checks* where certain relevant parameters are increased monotonically to see how running time and memory requirements scale. A simple example is the number of particles N in a n-body simulation. By calculating the running time τ for several small and manageable systems one can estimate $\tau \sim f(N)$ with $f(N)$ a function of the size of the input. If configurational averages are required—e.g., when performing disorder averages for spin glasses—these need to be factored into the total running time as well.

It is also useful to have the program produce periodic output steps during run time thus providing information on the progress of the simulation [e.g., via the built-in `time( )` function]. This has the advantage that further planning is possible during run time and the user has an idea on how much longer a simulation could take. A possible (machine-dependent output) to a logfile could be the header of the data files shown in Sec. 3, as well as the following useful information:

```
# process ID  :     7772
# hostname    :     moo.tamu.edu
# job start   :     Fri Apr 10 18:26:24 2009

# 2    samples:     50  % done
# elapsed time:     0 : 00 : 11.5
# timestamp   :     Fri Apr 10 18:26:37 2009

# 4    samples:     100 % done
# elapsed time:     0 : 00 : 23.1
# timestamp   :     Fri Apr 10 18:26:50 2009

# job end     :     Fri Apr 10 18:26:50 2009
```

The logged data show the process ID (PID) of the process in case it must be killed or paused, the hostname of the machine it is running on, as well as time-stamped progress reports on the data generation. Once the program exits, a final time stamp is printed. With this information, an arbitrary number of simulations on a large set of workstations can be managed easily, especially if they use a shared file system.

5.1 Managing simulations on large numbers of workstations

Some projects are *embarrassingly* parallel (this means that several simulations can be run in parallel without having them "talk" to each other) and thus can be simulated using workstation farms. Typical problems that fall into this category require configurational averages, i.e., the simulations have to be repeated many thousand times with either different parameters, data sets, or initial conditions. In general, it is recommended to use ready-set software such as Condor [12] which simplifies this task considerably by managing job submission and migration in case of a node failure. Most universities have at least one condor pool in an effort to tap into wasted cycles of desktop computers. The drawback of Condor is that only very specific compilers can be used and thus some codes either cannot be compiled at all or might not run at optimal speeds. Furthermore, the administrator configuration of condor can be cumbersome. By combining secure shell (`ssh` via OpenSSH) with some shell scripting, it is straightforward to obtain similar functionality.

Secure shell is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP ports can also be forwarded over the secure channel. Furthermore, by generating a keypair with `ssh-keygen` password-less logins can be performed securely. The reader is referred to the ssh manpages which contain a wealth of information. In this context only the basic functionality of ssh is presented within the context of managing several simulations on different workstations. It is assumed that the user has set up a ssh key pair to allow for password-less secure login.

Suppose that we have 3 computers, `moo`, `boo` and `goo` and we want to start simulations on them, manage the jobs, and possibly kill or pause them. A simple shell script wrapper (called `doo.sh`) which can be modified to accommodate all these operations is the following

```
#!/bin/sh

list="moo.tamu.edu boo.ethz.ch goo.ehtz.ch"

for mach in $list ;
do
    echo "$mach:"
    ssh $mach -n $1 $2 $3 $4 $5
    echo "============================================"
done
```

The first line calls the shell; in the third line we have a list of all machines we want to manage followed by a loop (lines 6 – 10) which calls the function `ssh $mach -n $1 ....` In the `ssh` command the option `-n` redirects standard input from `/dev/null`. The different `$i` take the options from the command line and pass them to the ssh command. In the following example we check the UTC date on the machines with the command `date -u`:

```
> ./doo.sh date -u
moo.tamu.edu:
Fri Apr 10 23:56:21 UTC 2009
=========================================================
boo.ethz.ch:
Fri Apr 10 23:56:24 UTC 2009
=========================================================
goo.ehtz.ch:
Fri Apr 10 23:56:28 UTC 2009
=========================================================
```

Therefore, any set of instructions can be distributed to a list of machines remotely. Clearly, using a shell script is rather rudimentary, and with the use of Perl, much fancier distribution methods can be accomplished. To start some jobs (`runme`) on the target machines in a directory called `run`, replace line 8 in `doo.sh` (`ssh $mach -n $1 $2...`) with

```
    ssh $mach -n "cd run; ./runme >& logfile &"
```

This changes the working directory to the directory `run` and executes the command `./runme >& logfile &`. To kill the jobs on all machines, use the original version of `doo.sh` in combination with Unix `killall` command:

```
> ./doo.sh killall runme
```

Finally, to check on the status of the (running) simulations, replace line 8 in `doo.sh` with

```
    ssh $mach -n "ps axo user,pid,%cpu,%mem,nice,stat,time,cmd \
                   \| egrep \"R \|RN \| RN+ \| R+ \""
```

Note that the command should be placed on one line and depends somewhat on the *nix flavor. Some testing with the `ps` command might be required. When run, it returns the status of the *running* programs only:

```
> ./doo.sh
moo.tamu.edu:
USER        PID %CPU %MEM  NI STAT     TIME CMD
hgk       19995 97.3  4.0  18 RN   20:43:59 ./runme
====================================================
boo.ethz.ch:
USER        PID %CPU %MEM  NI STAT     TIME CMD
hgk       32345 93.3  4.0  18 RN   20:32:49 ./runme
====================================================
goo.ehtz.ch:
USER        PID %CPU %MEM  NI STAT     TIME CMD
hgk       13455 95.3  4.0  18 RN   20:41:29 ./runme
====================================================
```

These are some simple tools that make managing jobs on up to approximately 100 workstations relatively straightforward.

5.2 HPC clusters

Explaining the details on how to run simulations on high-performance computer (HPC or supercomputer) clusters goes far beyond the scope of this lecture. Furthermore, there are many different ways to set up computer clusters and many different job schedulers (LSF, LoadLeveler, PBS, . . .). The latter are all similar, but unfortunately, the command line syntax is always different. There are, however, some basic steps that need to be performed (compilation and job submission) which are henceforth outlined. Probably the most important aspect to keep in mind is that programs should be thoroughly tested before being run on a cluster. First, wasting CPU time could be very costly, second, crashing the cluster might signify termination of the account.

Most clusters nowadays use `modules`. Because applications and compilers require several environmental variables, modules have been introduced. The idea is to call the necessary software via a module thus ensuring that the system configuration is properly set. For example, to check on the available modules and then load the Intel compiler type:

```
> module avail
> module load intel
```

After these commands have been issued, compilation with `icc` can be accomplished. The command `module help` lists all possible options. In general, jobs cannot be simply started, they need to be sent trough the *scheduler*. Using LSF, a bare-bones job submission can be accomplished via

```
> bsub -n nprocs -W hh:mm ./runme
```

Here nprocs is the number of requested processors and hh:mm represents the wall-clock time limit of the simulation in hours:minutes. Note: Only in *very* few very well managed clusters is this task as easy. In general, many more steps are required, that are thoroughly described in the documentation of the used scheduler. In the case of parallel simulations, compilation against MPI libraries [3] (or OpenMP) is required. Many supercomputer centers offer free online tutorials on code parallelization.

6 Libraries

Libraries are collections of routines and useful data types to be used in other programs. The idea is to have well-tested routines which are optimally programmed available for application development without having to program them from scratch. For example, we take for granted to use the sin( ) function which is actually part of the math library libm.a. Therefore, when we use sin( ) in a program, during compilation we need to link to the math library via -lm. In this section a quick Howto to generate libraries is given, followed by useful library packages for scientific computation.

6.1 Building libraries

Creating a library is a straightforward task: Suppose you have some programs that are generic and can be used by many of your applications, in this example myvecprod.c, mydate.c and myanalyze.c, together with a header file mytools.h which contains the necessary data types and function prototypes. The goal is to generate a library libmytools.a. First, object files need to be created:

```
> gcc -O2 -c myvecprod.c mydate.c myanalyze.c
```

Keep in mind that if you do not optimize the code on compilation the library calls will be slow! Therefore, as a reminder, in the previous example a -O2 flag has been added. The library is created with the Unix command ar

```
> ar r libmytools.a myvecprod.c mydate.c myanalyze.c
```

For further options of ar please look at the man page. After including new object files you have to update the internal object table of the library with

```
> ar s libmytools.a
```

Once the library has been created, suppose it is placed in /home/user/mytools/lib/ and the header in /home/user/mytools/include. To compile a program using routines from the header files and library, simply do

```
> gcc -o runme myprog.c -I/home/user/mytools/include \
       -L/home/user/mytools/lib/ -lmytools
```

By default, libraries are *dynamically* linked. This means that the code of the library is not included in your program and only called on run-time. This can be a problem if you use the program on other machines where the libraries are not installed. To overcome this problem *static* linking can be performed with the flag `-static` (or similar, see the compiler documentation) at the price of a larger executable.

6.2 Built-in C/C++ libraries

Standard C library The standard C library [13] is included by default on compilation. The only requirement is to include the header file `#include <stdlib.h>`. Most of the core operations such as memory allocation are included in this library. Therefore, its functions are very well documented in the man pages. For example, `man malloc` returns a detailed description of memory allocation functions, as well as their prototypes. There are many different header files in the standard library that can be included and thus expand the functionality of a program [13]. The most useful ones are

`stdio.h`
: Handles input/output of the program.

`math.h`
: Contains a multitude of useful mathematical functions (linking to the math libraries is needed via `-lm`).

`stdlib.h`
: Contains core functions for memory allocation, type conversion operators, (bad) random number generators, as well as routines for process control and interaction with the environment (e.g., system calls).

`time.h`
: Declares time and date functions that provide standardized access to time/date manipulation and formatting.

Standard template library (STL) The STL is part of the C++ Standard Library. It provides *containers* to store objects, *iterators* for container access, *algorithms*, and *functors* (function objects) [34]. It is far beyond the scope of the lecture to discuss the STL, the reader is referred to the documentation freely available on SGI's website [14].

6.3 Scientific libraries

GNU Scientific Library The GSL is a vast collection of routines aimed at solving numerical problems. It is freely available [15] and contains routines for the following numerical tasks:

- ☐ Complex number arithmetic

- ☐ Polynomials and root finding
- ☐ A vast list of special functions [21]
- ☐ Vector and matrix operations
- ☐ Interpolation, differentiation, integration
- ☐ Permutations and sorting
- ☐ Random number generation
- ☐ Statistical analysis
- ☐ Monte Carlo methods
- ☐ Fast Fourier transforms, etc.

Furthermore, the GSL interfaces with the BLAS (basic linear algebra subroutines in FORTRAN) which are the standard in linear algebra operations. To use the GSL, the necessary header files need to be included in the program. In the following example, the GSL is used to generate uniform random numbers:

```
#include <stdio.h>
#include <gsl/gsl_rng.h>

int main()
{
    gsl_rng *rng;                                 /* pointer to RNG */
    int     i;                                         /* iterator */
    int     n = 10;                     /* number of random numbers */
    double u;                                     /* random number */

    rng = gsl_rng_alloc(gsl_rng_mt19937); /* allocate generator */
    gsl_rng_set(rng,1234)                   /* seed the generator */

    for(i = 0; i < n; i++){
        u = gsl_rng_uniform(rng);    /* generate random numbers */
        printf("%f\n", u);
    }

    gsl_rng_free(rng);                         /* delete generator */

    return(0);
}
```

The numbers depend on the seed used by the generator. First the generator is allocated with `rng = gsl_rng_alloc(`*generator*`)` and seeded with `gsl_rng_set(rng,` *seed*`)`. Finally, the uniform random numbers in the interval $[0, 1[$ are produced with `gsl_rng_uniform(rng)`. Using the pre-defined GSL functions saves hours of programming and testing. The documentation of the GSL is extensive and usually has example problems in each section.

Numerical Recipes The Numerical Recipes [32] (NR) is a large collection of routines for many numerical problems (similar to the GSL) accompanied by a very useful book. The Numerical Recipes have been ported to different programming languages, such as C, C++, FORTRAN, FORTRAN90 and Pascal. The book used to be accessible online for free, but unfortunately this seems to have changed recently. The advantage over the GSL is that the routines have been ported to many programming languages, but the coding style is slightly outdated. Still, it is always a good idea to look at the NR book before you start any numerical project. Note that the single-user license allows (in theory) for only *one* executable to be run at a given point in time. Thus, to run software on multiple CPUs a group license is recommended.

Boost libraries The Boost libraries [16] provide free peer-reviewed portable C++ source libraries that extend the C++ STL. In order to ensure efficiency and flexibility, Boost makes extensive use of *templates*. Boost has been a source of extensive work and research into generic programming and metaprogramming in C++. The current Boost release contains approximately 80 individual libraries for a variety of tasks, such as

☐ Linear algebra

☐ Random number generation

☐ Regular expressions

☐ Graphs

☐ Math special functions and statistics

LEDA libraries LEDA [29] is a library dedicated to the efficient implementation of data types and algorithms. The library is commercial, although there is also a (crippled) free version [17]. Although LEDA is written in C++, it can also be accessed from standard C code. The free version includes the following data types:

☐ Data types for strings and multidimensional arrays

☐ Number data types (arbitrary precision)

☐ Stacks and queues, sets and trees

☐ Graphs (directed and undirected, labeled)

☐ Dictionaries (similar to Perl's hash)

☐ Data types for two- and three-dimensional geometries (points, spheres, etc.)

Furthermore, all necessary operations to manipulate standard data types are included. Finally, the library contains routines to compute strongly-connected components, shortest paths, maximum flows, minimum-cost flows, and minimum matchings for graphs, that are very useful when studying complex systems in statistical mechanics.

7 Data reduction and analysis with Perl

Perl is a high-level, general-purpose, interpreted, dynamic programming language. Perl is similar to C in syntax, but also borrows elements from shell scripting, awk and sed. Perl was originally developed to make report processing easier. Therefore, the language provides powerful text processing facilities without the arbitrary data length limits of many contemporary Unix tools, facilitating easy manipulation of text files. Because of its flexibility, Perl has been nicknamed "the Swiss Army chainsaw of programming languages." O'Reilly Media Inc. has published many books [1] about Perl, all of which are very useful.

It would be impossible to present the vast capabilities of Perl in this short lecture, therefore it is recommended that the reader looks at the literature [1]. To illustrate the use and simplicity of Perl, let us construct a few simple examples. Because Perl is a scripting language (although it can also be compiled), the typical syntax for a Perl script (called here `myscript.pl`) is

```
#!/usr/bin/perl -w
# this is a comment

print("hello world!\n");
```

On the command line, this script can then be executed with the Perl interpreter

```
> perl myscript.pl
hello world!
```

The Perl syntax is very flexible, for example, `print("hello world!\n");` could be replaced by `print "hello world!\n";`. Each statement is terminated by a semi-colon, similar to other high-level languages. Suppose we have a set of *several* files which contain tabulated numbers (see Sec. 3 for an example) and we want to compute an average over the third column in the files. Here is a Perl script to accomplish this task (called `average.pl`):

```
#!/usr/bin/perl -w

if($#ARGV < 0){
    warn "\n\toops! type $0 files.\n"; exit;
}
else{
    @files = @ARGV;
}

@table   = ();            # array where data are stored
$counter = 0;             # counter

foreach $file (@files){   # loop over files
    open(DATA, "$file");  # open file
15      while (<DATA>){         # parse file
16          chomp;              # remove newlines
17          if ( $_ =~ /^#/){ # skip comment lines in files
18              next;
19          }
20          $table[$counter] = (split /\s+/)[2];
21          $counter++;
22      }
23      close(DATA);
24  }
25
26  $mean = 0;
27  for($i = 0; $i < $counter; $i++){
28      $mean += $table[$i];
29  }
30  $mean /= $counter;
31
32  printf("mean = %4.3f\n",$mean);
```

Running the perl script on multiple data files on the command line one obtains:

```
 > perl average.pl *.dat
 77.72
```

Let us dissect `average.pl` and highlight the advantages of using the Perl programming language. In line 1 the Perl interpreter is called. The option `-w` turns on warnings which, in general, is useful to do when debugging. From lines 3 – 8 we first check if files have been given as arguments to the script. If not, we exit with an error message. If yes, the file names are stored in an array `@files`. In Perl, arrays always start with "`@`." Variables usually start with a `$`, such as in the initialization of `$counter`. The array `@table` is initialized to be empty. One useful function in Perl is the `foreach( )` function. In this case it allows us to loop over all file names in `@files` and manipulate them. A data stream is opened in line 14. While the data are being parsed line by line, newlines at the end of the lines are removed with `chomp` in line 16. Line 17 highlights the first major strength of Perl: full support of regular expressions. We check that the currently parsed line (`$_`) does not start with a hash by matching the string `/^#/` using the matching operator `~`. If it does, we skip the line (`next`). In line 20 the data on each line of the file are split by (regexp) matching white spaces and we select the number in the third column (indices start with zero) to be stored in `$table[$counter]`. Once the file has been parsed, the data stream is closed. After all files in `@files` have been processed, we compute—in this simple example—the mean of the numbers.

Clearly, there are far more elegant ways of performing this task in Perl. In an attempt to keep the syntax simple, readable, and as close as possible to C, we have written the program in this way. Furthermore, performing an average is a trivial task that could be performed on the command line with the `cat` and `awk` commands. Far more complex data operations can be performed once the data are read into memory. Further features of Perl:

☐ In addition to the variable and array data types, Perl also allows *hashes* which can be seen as *generalized* arrays where the *indices* can be arbitrary quantities and do not have to be numbers. Example: `%kitchen = (apple => 'red', banana => 'yellow')`. Calling `$kitchen{apple}` returns `red`.

☐ Support for subroutines.

☐ Full support for regular expression pattern matching.

☐ Perl *modules* from CPAN [18]. These are similar to libraries in C. CPAN provides a huge list of modules for almost any thinkable task. For example, interfaces to eBay, image manipulation, server and demon utilities, Boost data types, to name a few. These can be installed and then called from Perl scripts.

☐ Full portability of the code. Since the program is interpreted on the fly and does not require compilation, Perl scripts can be run on many different platforms and architectures, as long as the interpreter is installed.

Although Perl is not very fast, the ease of use (especially for C programmers) makes it the language of choice for data post processing.

8 Data visualization and fitting: gnuplot

Once the simulation data have been produced and reduced, analysis and visualization need to be performed. There are many software packages that accomplish these tasks, each with their strengths and weaknesses. For example, data plotting can be accomplished with xmgrace (GUI plotting application), gnuplot (command line), SuperMongo (scripting language) or Mathematica. One (freely-available) software package which has good data fitting capabilities and can produce decent-looking figures is gnuplot [19]. In this section the core functionality of gnuplot is presented. Gnuplot can be started on any *nix system via

```
> gnuplot
```

Note that gnuplot can also be used in batch mode (running scripts without direct users interaction). This is useful for interfacing with data analysis tools, as well as plotting many images. Gnuplot has an extensive built-in help system, which can be called via the `help` command. `help help` gives an introduction to the built-in help.

8.1 Plotting data

Gnuplot can be used both to do quick-and-dirty plots of data to obtain an impression on how things look, as well as publication-quality figures.

Quick-and-dirty In general, for quickly displaying tabulated data of the form `x y err_y` stored in a file `results.dat` the following commands have to be issued at the gnuplot prompt:

```
gnuplot> plot [0:3][-2:10] "results.dat" with yerrorbars
```

The previous command plots the numbers in `results.dat` in the `x`-range $[0, 3]$ and the `y`-range $[-2, 10]$ with error bars. Note that '`with yerrorbars`' can also be abbreviated with '`w e`.' A X11 window pops up displaying the data (see Fig. 1). Placing the cursor *over* the X11 window with the data and hitting the `h`-key lists all possible *interactive* commands in the terminal window. For example, hitting the `L`-key close to an axis near the cursor changes the axis from linear to logarithmic. Finally, to see all plotting options as far as line styles and symbols is concerned, simply enter the command `test` in the command line.

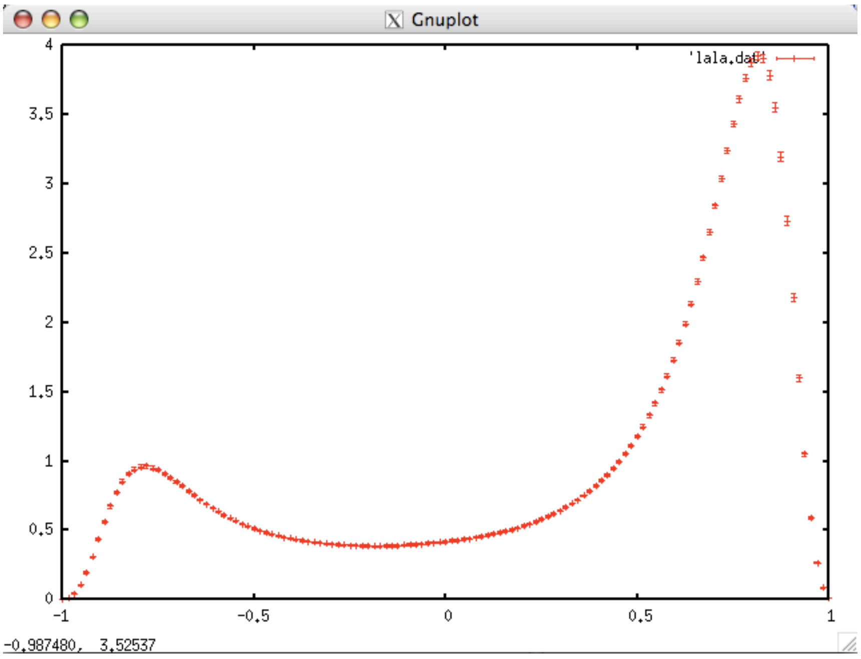

Figure 1: Sample X11 terminal output from gnuplot for a quick-and-dirty plot.

Publication-quality plots To generate "presentable" plots of data, it is highly recommended to write a gnuplot script (`macro.gp`) and generate an encapsulated PostScript file in batch mode:

```
 > gnuplot macro.gp
```

The reason is that many options need to be set and the figure tweaked to ensure that it is nice and fits the journal format. The following script produces publication-quality images. Note that these can and must be further improved. This is merely a proof-of-concept macro:

```
 1 # preamble (set parameters)
 2 set size ratio 1 2,2
 3 set terminal postscript color eps enhanced "Times-Roman" 48
 4 set output "results.eps"
 5 set origin 0,0
 6 unset label
 7 set size square
 8 set pointsize 1.5
 9 set key 0.3, 3.5
10
11 # x-axis tick, ranges and labels
12 set xrange [-1.0:1.0]
13 set xtics -1.0,0.5,1.00
14 set mxtics 5
15 set xlabel "{/Symbol l}"
16
17 # y-axis ticks, ranges and labels
18 set yrange [0.0:4.0]
19 set ytics 0,1,4
20 set mytics 5
21 set ylabel "{/Times-Italic P}_{c}({/Symbol l})"
22
23 # add other text labels
24 set label "(b)" at -0.75, 3.50
25 set label "{/Times-Italic T} = 0.20" at 0.4, 0.5
26
27 # plot points with errors, draw line trough them
28 set multiplot
29 plot  "results.dat" using 1:2:3 with yerrorbars pointtype 5 \
30         ti "{/Times-Italic N} = 128"
31 plot  "results.dat" using 1:2 with lines notitle lt -1
32 unset multiplot
33
34 # fix bounding box with perl
35 !perl -pi -e ’s/%%BoundingBox: 50 50 770 554/%%BoundingBox: 50 50 560 560/’\
36         results.eps
```

The most relevant lines and commands are explained below. For further information on the different commands used, the reader is referred to the gnuplot documentation or the Howto by T. Kawano [20]. Line 3 determines the `terminal` to use. Default is X11 (display to the screen), in this case we change it to produce an encapsulated

PostScript file with a `Times-Roman` font of 48 points and the file name "`results.eps`" (line 4). The blocks starting at lines 11 and 17 determine the look and labels of the axes. Note that specific fonts can be called to render the labels, including the Greek alphabet (see Ref. [20] for details). Further text labels are added after line 23. In line 28 gnuplot is instructed to plot multiple data sets. The reason is that in line 29 we plot the data points as red squares with error bars, and in line 31 we plot the same data again connecting the points, but without symbols. Finally, in lines 35/36 we use a system call to the shell invoking perl to change the bounding box of the resulting encapsulated PostScript such that the figure is square (better for two-column formatted journals). The output is shown in Fig. 2 and is clearly superior than the plot shown in Fig. 1.

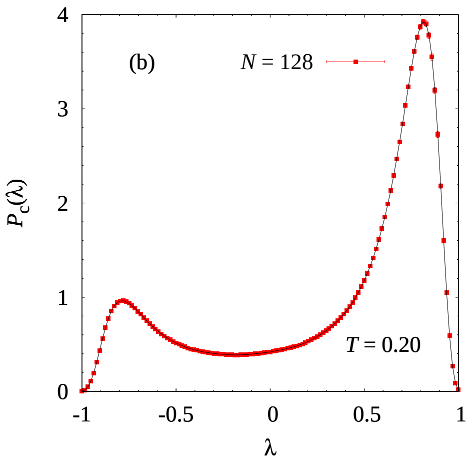

Figure 2: Print-quality figure generated with the presented gnuplot macro. It shows the same data as in Fig. 1. The aspect ratio has been set to unity to better fit a two-column format. This can be undone by removing the line '`set size square`' in the macro.

8.2 Fitting data

Gnuplot also offers many options for data manipulation. A very useful one is data fitting. Suppose we want to fit some data file of the form `x y err_y` to an exponential decay of the form $f(x) = a\exp(-bx)$ with a and b parameters. At the `gnuplot` prompt we first enter the function followed by some starting values. The starting values are "good guesses" of what the parameters should be to help the Levenberg-Marquardt nonlinear least-squares fitting routine converge.

```
gnuplot> f(x) = a*exp(-b*x)
gnuplot> a = 100.0
gnuplot> b = 2.0
```

The fit to data in a file `results.dat` is then performed using the `fit` function

```
gnuplot> fit f(x) "results.dat" using 1:2:3 via a,b
```

The subcommand "`using 1:2:3 via a,b`" tells gnuplot to use the first, second and third column for `x y err_y`, respectively and to fit with a and b as parameters. We could also, for example, fix a and only vary b in the fit by changing "`via a,b`" to "`via b`." Gnuplot then attempts to fit the curve to the data and outputs a log of the iterative process, as well as the optimal parameters (with error bars) and the χ^2 per degree of freedom of the fit [32] (measure of the quality of the fit). Furthermore, a correlation matrix between the parameters is provided. Typical (abridged) output:

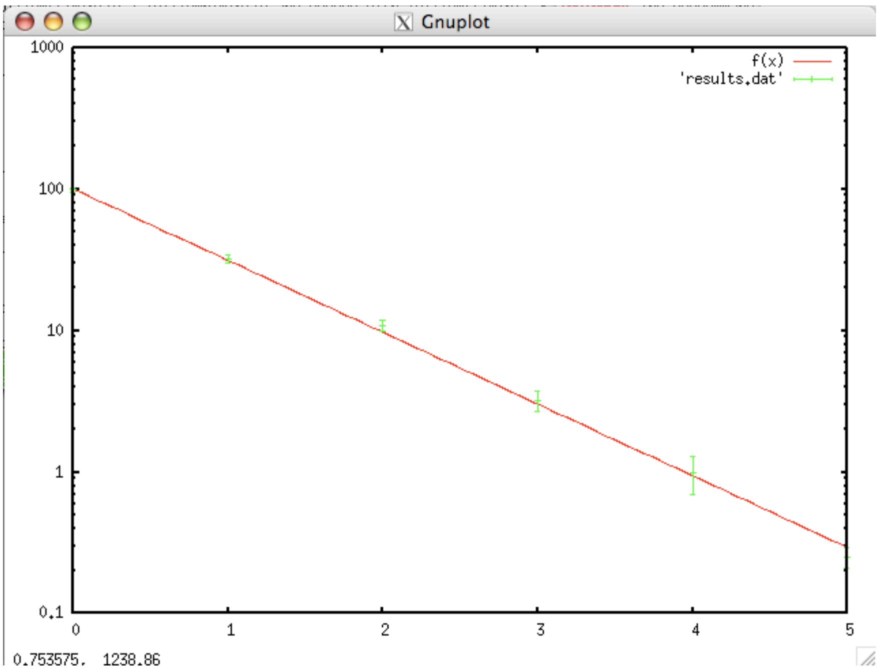

Figure 3: Visual verification that the fitting routine has converged to the correct minimum. The fit (solid line) follows the data points closely.

```
After 5 iterations the fit converged.
final sum of squares of residuals : 3.09786
rel. change during last iteration : -2.86707e-16

degrees of freedom (ndf) : 4
rms of residuals      (stdfit) = sqrt(WSSR/ndf)      : 0.880036
variance of residuals (reduced chisquare) = WSSR/ndf : 0.774464

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 100.219          +/- 2.486        (2.481%)
b               = 1.16714          +/- 0.01981      (1.698%)

correlation matrix of the fit parameters:

                a      b
a               1.000
b               0.436  1.000
```

In this case the optimal parameters are $a = 100.2 \pm 2.5$ and $b = 1.167 \pm 0.019$. One should always also verify the quality of the fit visually via

```
gnuplot> plot f(x), "results.dat" w e
```

In this case, the fitting routine seems to have converged to the proper minimum, as can be seen in Fig. 3 (linear-log plot).

If the fit does not seem to agree with the data or it does not converge, retry the fitting procedure with new starting values.

Disclaimer & Acknowledgments

I would like to thank Juan Carlos Andresen and Ruben Andrist for critically reading the manuscript, as well as John Chang for numerous suggestions on the first edition of the lecture notes.

Finally, I would like to add the disclaimer that the presented approaches are by no means optimal for simulating all possible scientific problems. They definitely do not scale to large software projects with many collaborators and they are purely based on the author's own experiences in simulating physical systems. But they *do* provide a starting point to tackle problems using numerical methods.

References

[1] http://www.oreilly.com.

[2] http://www.gnu.org/software/bash.

[3] http://www.mcs.anl.gov/research/projects/mpi.

[4] http://en.wikipedia.org/wiki/Regular_expression.

[5] http://www.eclipse.org.

[6] http://developer.apple.com/TOOLS/xcode.

[7] http://subversion.tigris.org.

[8] http://mercurial.selenic.com.

[9] http://www.hdfgroup.org/HDF5.

[10] http://www/valgrind.org.

[11] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[12] http://www.cs.wisc.edu/condor.

[13] http://en.wikipedia.org/wiki/C_standard_library.

[14] http://www.sgi.com/tech/stl.

[15] http://www.gnu.org/software/gsl.

[16] http://www.boost.org.

[17] http://www.algorithmic-solutions.com/leda.

[18] http://www.cpan.org.

[19] http://www.gnuplot.info.

[20] http://t16web.lanl.gov/Kawano/gnuplot/index-e.html.

[21] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, New York, 1964.

[22] K. Binder and A. P. Young. Spin glasses: Experimental facts, theoretical concepts and open questions. *Rev. Mod. Phys.*, 58:801, 1986.

[23] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion.* O'Reilly Media Inc., Sebastopol, 2008. (http://svnbook.red-bean.com/).

[24] H. T. Diep. *Frustrated Spin Systems.* World Scientific, Singapore, 2005.

[25] A. K. Hartmann and H. Rieger. *Optimization Algorithms in Physics.* Wiley-VCH, Berlin, 2001.

[26] E. Ising. Beitrag zur Theorie des Ferromagnetismus. *Z. Phys.*, 31:253, 1925.

[27] M. Loukides and A. Oram. *Programming with GNU Software.* O'Reilly Media Inc., Sebastopol, 1997.

[28] S. W. Means and E. R. Harold. *XML in a Nutshell.* O'Reilly Media Inc., Sebastopol, 2001.

[29] K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, 1999. also `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[30] M. Mézard, G. Parisi, and M. A. Virasoro. *Spin Glass Theory and Beyond.* World Scientific, Singapore, 1987.

[31] A. Oram and S. Talbott. *Managing Projects with Make.* O'Reilly Media Inc., Sebastopol, 1991.

[32] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 1995.

[33] I. Sommerville. *Software Engineering.* Addison-Welsey, Reading, 1989.

[34] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Redwood City, 2000.

[35] J. M. Yeomans. *Statistical Mechanics of Phase Transitions.* Oxford University Press, Oxford, 1992.