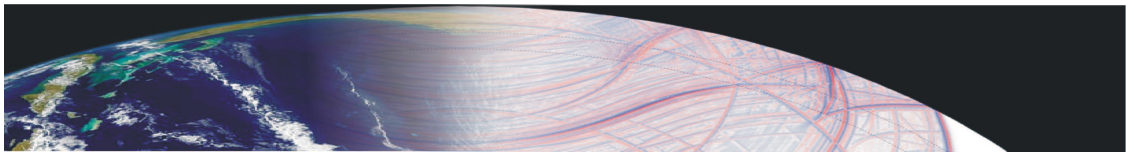

SPICE library of codes: Recommendations

Department of Earth and Environmental Sciences – Geophysics Section
SPICE Research Training Network – www.spice-rtn.org

Preface

In August 2004 the Munich Geophysics Section organized an open SPICE-funded "brainstorming" weekend with the goal of developing some fundamental guidelines that we considers as important for applications that may enter a code library. This document contains part of the results of this meeting. The idea is that this is a living document that is adapted with the future developments and comments by others. In the future, we hope to (1) standardize the way our codes are structured, (2) to have common input and output formats, (3) to share libraries and tools for visualization and processing, and finally and most importantly (4) spend more time doing science!

Note that the recommendations given here serve merely as suggestions! The most important aspect, however, is that there is **sufficient documentation** accompanying the codes, making them useful to others.

Thanks to the participants Bernhard Schuberth, Erika Vye, Giampiero Iaffaldano, Gilbert Brietzke, Gunnar Jahnke, Haijiang Wang, Jens Oeser, Josep de la Puente, Marco Stupazzini, Markus Treml, Martin Käser, Michael Ewald, Peter Bunge, Peter Danecek, Richard Bennett, Robert Barsch, Steffi Rieger, Teresa Reinwald, Toni Kraft, Wiwit Suryanto and particularly Dr. Bader (LRZ Munich) for their suggestions and contributions.

Heiner Igel, Munich, June 2005

Recommended literature (Fortran)

- **Fortran 95/2003 explained,** Michael Metcalf, Oxford University Press
- **Numerical Recipes in Fortran 90,** Brian P. Flannery, 15 January, 1996 Cambridge University Press
- **Numerical Recipes in FORTRAN Example Book :** The Art of Scientific Computing, William T. Vetterling, 27 November, 1992 Cambridge University Press, Paperback - Show all editions
- **Classical Fortran,** Michael Kupferschmid, Dekker

File Structure

When developing codes that may be used by others in your group or outside it makes sense to define a common file structure that one immediately can recognize. The following structure is recommended:

/main (program name)	**/src**		
		/obj	**/arch1**
			/arch2
		/bin	**/arch1**
			/arch2
	/doc		
	/examples		
	/output		
	/tools		

Description of file structure:

- /**main** contains config file for generation of machine specific makefiles, or general makefile, readme file, etc.
- /**src** contains the makefile to compile the code(s)
- /**obj/arch1,2..** contains object files generated during compile time for various hardware architectures
- /**bin/arch1,2..** contains executables for various hardware architectures
- /**doc** contains all documentation materials, user's guide, supporting material (papers), readme files, etc.
- /**output** directory for any data output by the programs
- /**tools** visualisation tools, processing programs, etc.
- /**examples** parameter files with examples (e.g. recovery of analytical solutions, known benchmarks)

Makefiles

Ideally the source code can be compiled in a straight forward way by typing the unix/linux command "make" in the appropriate directory, that should be given in the documentation. Make then runs the commands given in the *Makefile.* This facilitates the compiling and linking of programs, particularly if one is combining several source files and links additional libraries. Refer to any Unix/Linux manuals for more details. Features that may be part of a makefile are:

- location of src (obj,exe) files
- optimization flags
- libraries
- architecture specific flags
- parallelization specific flags
- debug flags
- etc.

Makefiles can be made arbitrarily complex. However, they should be made as self-explanatory as possible and the area where things can and should be changed by potential users should be clearly identified.

Parameter Input and Output

Most application programs are taking in information from a parameter file either of a fixed name, or as standard input that is piped in (e.g. program.x < Par). The advantage is that programs do not need to be recompiled before execution. It is therefore useful to define a common format to do this, even though it is a matter of taste. It is worth noting that an "as free as possible" input data file format is highly recommended (e.g.: avoid fixed dimension for tabs and spacing between input parameters or any arbitrary input sequence if it is not important for the simulation); this recommendation tends to prevent user mistakes due to a complex and highly constrained data structure.

Below an example of a parameter file is given. The Fortran program reads either a junk string character for the comment lines or a combination of a string and a variable, in case there is a variable to read. The input format has to be specified explicitly. It is useful, if documentation/description of each parameter is given in the parameter file (as well as in the readme file/user's guide etc.).

```
Sample Parameter file  ==================
seisfile      =DATA/test                  !name of simulation run
nt            =500                        !number of times steps
dt            => from stability criterion
Model ------------------------
model_type    =1                          !fork for model initialization
vs0           =2886.75134                 !vs for homo model
ixa           =5                          !first record in x
ixe           =45                         !last record in x
……
```

Probably one of the most difficult issues is how to initialise (seismic) model parameters. As this will strongly depend on the particular methods used we will not make any recommendations here. The parameter files are often of fundamental importance because it may be the only part of a program potential users will modify. Therefore it is highly recommended that they contain as much information on the variables as somehow possible (see also Documentation section).

In order to post-process our data, store details about particular runs etc. current programs may output a substantial amount of data. We may distinguish between

general type of information on the particular run (parameters) and actual simulation results (e.g. snapshots, seismograms, complete 3D fields, etc.).

Logfiles: During run time the actual parameters that are used in the simulation should be output to a log file, either by redirecting standard output (e.g. *program.x < Par > log)* or though other means. This helps enormously if things go wrong and also allows later to reproduce results. If the simulation is highly CPU time consuming (days or weeks) it is worth introducing into the code a "restore option" at certain fixed time of the analysis. In this way if the analysis crashes (e.g.: blackout or hardware failure) user is not obliged to run once again the complete simulation.

Consistency checks: Good codes perform a substantial amount of consistency checks before they start the actual work. Programs should stop (possibly with an appropriate statement about what went wrong) whenever there is some inconsistency between simulations, physical, etc. parameters leading to wrong results (maybe hidden, maybe at the end of long and expensive simulations, etc.). In numerical wave propagation codes this applies particularly to stability problems.

Other output: It makes sense to output various types of information in separate files and give those files sensible names, e.g.:

- name_of_run_src (for a source time function)
- name_of_run_log (log file)
- name_of_run_receivers (for receiver locations)
- etc.

Simulation results: Clearly the types of data to be stored for a simulation may vary strongly. However, if programs are to be made available for others, it is recommended that – even though slow and memory intensive – ASCII output is give as an option, as this is the easiest format to reprocess. A further option should be the output as "unformatted" format, which is a lot faster and requires less storage. Some hardware incompatibilities are possible, however.

As far as seismic simulation data are concerned, it is desirable to have an option for output of seismograms for example as SAC or miniSEED format. This allows inputting the results directly into standard seismic software packages for further processing.

Documentation

By documentation we mean here the information that is provided with the code if it is distributed to other users. The golden rule is: **PROVIDE AS MUCH INFORMATION AS POSSIBLE!** We distinguish here between internal (inside the code) and external (other) documentation.

Internal

Some simple rules may help making a code more understandable:

- provide introductory description for each subroutine or subprogram
- use some standard way for documenting code changes or bug fixes
- use self-explanatory name conventions in combination with module names
- don't use cryptic subroutine names, give names that explain their function

- if the code is shared by a team it is worth introducing *tag* easily recognizable developers and self-explanatory about *when*, *who* and *why* a particular line was introduced or modified.

External

External documentation should be as extensive as possible, the minimum is the provision of a *README* file that contains all required information to get started with the code. Ideally – with increasing number of users – a README file should transform into a "User's Guide". The following information MUST be provided with each code:

- authors
- history (in case it is based on previous codes)
- references (any literature that helps in understanding the code)
- date/version number (in case version control is used)
- bug fixes
- e-mail address for support
- licensing info
- FAQs
- Getting started (simple instructions to get results)
- Hardware limitations, platforms on which code is tested
- Decription of file structure, file contents
- Description of any auxiliaries
- Flow charts, data structure (e.g. graphically), use for example Totalview output
- Release notes
- Provide Parameter file that reproduces known solution (e.g. analytical solution) and provide the required data to compare with
- HOW TO (ppt or pdf format) with same reference case completely developed and explained

Parallellization

Most simulation programs in geophysics will necessitate the use of the parallel programming paradigm for some time to come. Ideally programs should be provided in **serial** and in **parallel** form using the MPI language. One way to avoid having to develop two codes in parallel is to use a library with dummy MPI subroutines, that can be used when the code runs in serial on a machine that may not have the MPI libraries. In this case the specific library could be linked and the program runs on a single processor.

Please consult any of the MPI courses that are available online for any information on how to parallelize FORTRAN code.

Version Control

Version control is a formal way of keeping track of changes that are being made to any type of document of program. A common program that is used is **CVS** or **subversion**. Formal version control is a MUST if more than one person is developing the code, but is also highly recommended for single-user development.

License issues and access

The codes that SPICE will administer through their www pages will be distributed under some open source lices (e.g., the gnu public license GPL - http://www.gnu.org). To download software users need to register and accept an agreement that (1) the software is not used for commercial applications, (2) that there is no warranty whatsoever as to the correctness of the codes, and (3) that the codes' authors remain the contact persons for any further information. These items are still subject to discussions. But in any case some sort of agreement should be part of the download process.