Reproducible computational experiments using SCons

Sergey Fomel[1] *and Gilles Hennenfent*[2]

ABSTRACT

SCons (from Software Construction) is a well-known open-source program designed primarily for building software. In this paper, we describe our method of extending SCons for managing data processing flows and reproducible computational experiments. We demonstrate our usage of SCons with a couple of simple examples.

INTRODUCTION

This paper introduces an environment for reproducible computational experiments developed as part of the "Madagascar" software package. To reproduce the example experiments in this paper, you can download Madagascar from `http://www.ahay.org/`. At the moment, the main Madagascar interface is the Unix shell command line so that you will need a Unix/POSIX system (Linux, Mac OS X, Solaris, etc.) or Unix emulation under Windows (Cygwin, SFU, etc.)

Our focus, however, is not only on particular tools we use in our research but also on the general philosophy of reproducible computations.

Reproducible research philosophy

Peer review is the backbone of scientific progress. From the ancient alchemists, who worked in secret on magic solutions to insolvable problems, the modern science has come a long way to become a social enterprise, where hypotheses, theories, and experimental results are openly published and verified by the community. By reproducing and verifying previously published research, a researcher can take new steps to advance the progress of science.

Traditionally, scientific disciplines are divided into theoretical and experimental studies. Reproduction and verification of theoretical results usually requires only imagination (apart from pencils and paper), experimental results are verified in laboratories using equipment and materials similar to those described in the publication.

[1]University of Texas at Austin, E-mail: sergey.fomel@beg.utexas.edu
[2]Earth & Ocean Sciences, University of British Columbia, E-mail: ghennenfent@eos.ubc.ca

During the last century, computational studies emerged as a new scientific discipline. Computational experiments are carried out on a computer by applying numerical algorithms to digital data. How reproducible are such experiments? On one hand, reproducing the result of a numerical experiment is a difficult undertaking. The reader needs to have access to precisely the same kind of input data, software and hardware as the author of the publication in order to reproduce the published result. It is often difficult or impossible to provide detailed specifications for these components. On the other hand, basic computational system components such as operating systems and file formats are getting increasingly standardized, and new components can be shared in principle because they simply represent digital information transferable over the Internet.

The practice of software sharing has fueled the miraculously efficient development of Linux, Apache, and many other open-source software projects. Its proponents often refer to this ideology as an analog of the scientific peer review tradition. Eric Raymond, a well-known open-source advocate, writes (Raymond, 2004):

> Abandoning the habit of secrecy in favor of process transparency and peer review was the crucial step by which alchemy became chemistry. In the same way, it is beginning to appear that open-source development may signal the long-awaited maturation of software development as a discipline.

While software development is trying to imitate science, computational science needs to borrow from the open-source model in order to sustain itself as a fully scientific discipline. In words of Randy LeVeque, a prominent mathematician (LeVeque, 2006),

> Within the world of science, computation is now rightly seen as a third vertex of a triangle complementing experiment and theory. However, as it is now often practiced, one can make a good case that computing is the last refuge of the scientific scoundrel [...] Where else in science can one get away with publishing observations that are claimed to prove a theory or illustrate the success of a technique without having to give a careful description of the methods used, in sufficient detail that others can attempt to repeat the experiment? [...] Scientific and mathematical journals are filled with pretty pictures these days of computational experiments that the reader has no hope of repeating. Even brilliant and well intentioned computational scientists often do a poor job of presenting their work in a reproducible manner. The methods are often very vaguely defined, and even if they are carefully defined, they would normally have to be implemented from scratch by the reader in order to test them.

In computer science, the concept of publishing and explaining computer programs goes back to the idea of *literate programming* promoted by Knuth (1984) and expended by many other researchers (Thimbleby, 2003). In his 2004 lecture on "better

programming", Harold Thimbleby notes[1]

> We want ideas, and in particular programs, that work in one place to work elsewhere. One form of objectivity is that published science must work elsewhere than just in the author's laboratory or even just in the author's imagination; this requirement is called *reproducibility*.

Nearly ten years ago, the technology of reproducible research in geophysics was pioneered by Jon Claerbout and his students at the Stanford Exploration Project (SEP). SEP's system of reproducible research requires the author of a publication to document creation of numerical results from the input data and software sources to let others test and verify the result reproducibility (Claerbout, 1992a; Schwab et al., 2000). The discipline of reproducible research was also adopted and popularized in the statistics and wavelet theory community by Buckheit and Donoho (1995). It is referenced in several popular wavelet theory books (Hubbard, 1998; Mallat, 1999). Pledges for reproducible research appear nowadays in fields as diverse as bioinformatics (Gentleman et al., 2004), geoinformatics (Bivand, 2006), and computational wave propagation (LeVeque, 2006). However, the adoption or reproducible research practice by computational scientists has been slow. Partially, this is caused by difficult and inadequate tools.

Tools for reproducible research

The reproducible research system developed at Stanford is based on "make (Stallman et al., 2004)", a Unix software construction utility. Originally, SEP used "cake", a dialect of "make" (Nichols and Cole, 1989; Claerbout and Nichols, 1990; Claerbout, 1992b; Claerbout and Karrenbach, 1993). The system was converted to "GNU make", a more standard dialect, by Schwab and Schroeder (1995). The "make" program keeps track of dependencies between different components of the system and the software construction targets, which, in the case of a reproducible research system, turn into figures and manuscripts. The targets and commands for their construction are specified by the author in "makefiles", which serve as databases for defining source and target dependencies. A dependency-based system leads to rapid development, because when one of the sources changes, only parts that depend on this source get recomputed. Buckheit and Donoho (1995) based their system on MATLAB, a popular integrated development environment produced by MathWorks (Sigmon and Davis, 2001). While MATLAB is an adequate tool for prototyping numerical algorithms, it may not be sufficient for large-scale computations typical for many applications in computational geophysics.

"Make" is an extremely useful utility employed by thousands of software development projects. Unfortunately, it is not well designed from the user experience

[1] `http://www.uclic.ucl.ac.uk/harold/`

prospective. "Make" employs an obscure and limited special language (a mixture of Unix shell commands and special-purpose commands), which often appears confusing to unexperienced users. According to Peter van der Linden, a software expert from Sun Microsystems (van der Linden, 1994),

> "Sendmail" and "make" are two well known programs that are pretty widely regarded as originally being debugged into existence. That's why their command languages are so poorly thought out and difficult to learn. It's not just you – everyone finds them troublesome.

The inconvenience of "make" command language is also in its limited capabilities. The reproducible research system developed by Schwab et al. (2000) includes not only custom "make" rules but also an obscure and hardly portable agglomeration of shell and Perl scripts that extend "make" (Fomel et al., 1997).

Several alternative systems for dependency-checking software construction have been developed in recent years. One of the most promising new tools is SCons, enthusiastically endorsed by Dubois (2003). The SCons initial design won the Software Carpentry competition sponsored by Los Alamos National Laboratory in 2000 in the category of "a dependency management tool to replace make". Some of the main advantages of SCons are:

- SCons configuration files are Python scripts. Python is a modern programming language praised for its readability, elegance, simplicity, and power (Rossum, 2000a,b). Scales and Ecke (2002) recommend Python as the first programming language for geophysics students.
- SCons offers reliable, automatic, and extensible dependency analysis and creates a global view of all dependencies – no more "make depend", "make clean", or multiple build passes of touching and reordering targets to get all of the dependencies.
- SCons has built-in support for many programming languages and systems: C, C++, Fortran, Java, LaTeX, and others.
- While "make" relies on timestamps for detecting file changes (creating numerous problems on platforms with different system clocks), SCons uses by default a more reliable detection mechanism employing MD5 signatures. It can detect changes not only in files but also in commands used to build them.
- SCons provides integrated support for parallel builds.
- SCons provides configuration support analogous to the "autoconf" utility for testing the environment on different platforms.
- SCons is designed from the ground up as a cross-platform tool. It is known to work equally well on both POSIX systems (Linux, Mac OS X, Solaris, etc.) and Windows.

- The stability of SCons is assured by an incremental development methodology utilizing comprehensive regression tests.
- SCons is publicly released under a liberal open-source license[2]

In this paper, we propose to adopt SCons as a new platform for reproducible research in scientific computing.

Paper organization

We first give a brief overview of "Madagascar" software package and define the different levels of user interactions. To demonstrate our adoption of SCons for reproducible research, we then describe a couple of simple examples of computational experiments and finally show how SCons helps us document our computational results.

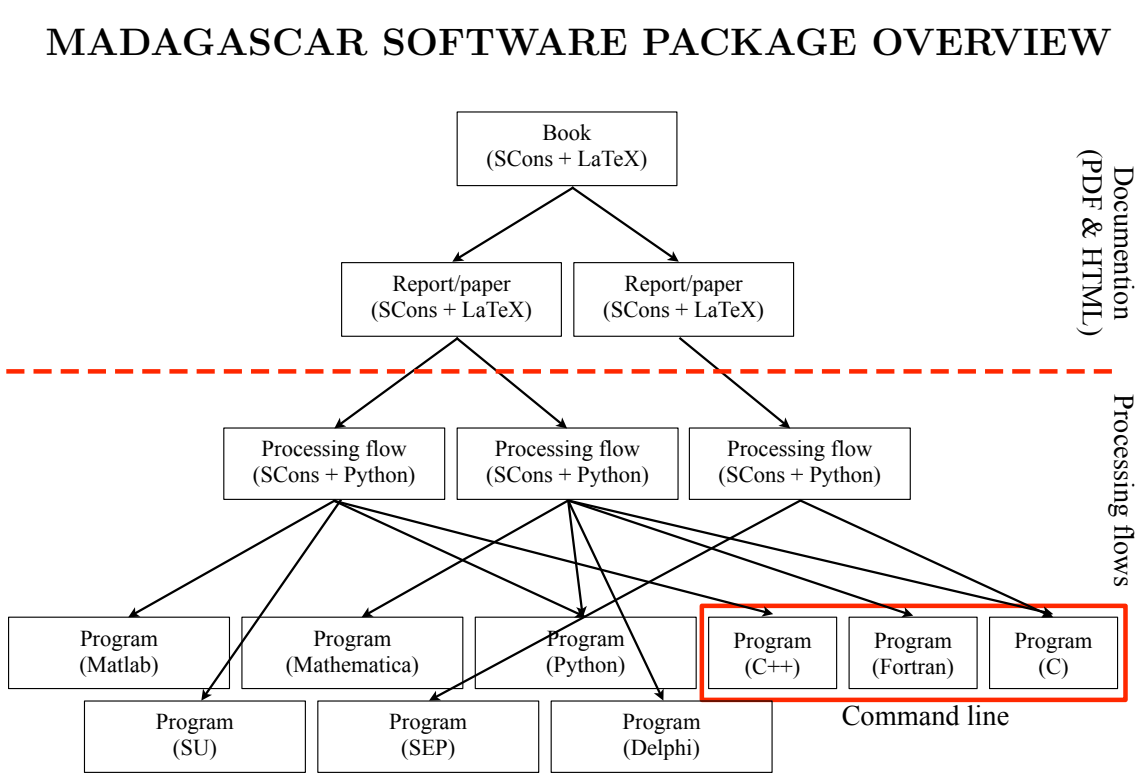

Figure 1: caption

"Madagascar" is a multi-layered software package (Fig. 1). Users can thus use it in different ways:

[2]As of time of this writing, SCons is in a beta version 0.96 approaching the 1.0 official release. See `http://www.scons.org/`.

- **command line**: "Madagascar" is first of all a collection of command line programs. Most programs act as filters on input data and can be chained in a Unix pipeline, e.g.

  ```
  sfspike n1=200 n2=50 | sfnoise rep=y >noise.rsf
  ```

  Although these programs mainly focus at this point on geophysical applications, users can use the API (application programmer's interface) for writing their own software to manipulate Regularly Sampled Format (RSF) files, "Madagascar" file format. The main software language of "Madagascar" is C. Interfaces to other languages (C++, Fortran-77, Fortran-90, Python) are also provided.

- **processing flows**: "Madagascar" is also an environment for reproducible numerical experiments in a very broad sense. These numerical experiments (or "computational recipes") can be done not only using "Madagascar" command line programs but also Matlab, Mathematica, Python, or other seismic packages (e.g. SEP, Seismic Unix). We adopted SCons for this part as we shall demonstrate later.

- **documentation**: the most upper layer of "Madagascar" and maybe the most critical for reproducible research is documentation. "Madagascar" establishes a direct link between the figures of a paper or a report and the codes that were used to generate them. This layer uses SCons in combination with LaTeX to generate PDF, HTML, and MediaWiki files real easy and undoubtly makes "Madagascar" an environment of choice for technology transfer, report, thesis, and peer-reviewed publication writing.

EXAMPLE EXPERIMENTS

The main `SConstruct` commands defined in our reproducible research environment are collected in Table 1.

These commands are defined in `$RSFROOT/lib/rsfproj.py` where `RSFROOT` is the environmental variable to the Madagascar installation directory. The source of this file is in python/rsfproj.py.

Example 1

To follow the first example, select a working project directory and copy the following code to a file named `SConstruct`[3].

```
1 from rsf.proj import *
2
```

[3]The source of this file is also accessible at book/rsf/scons/easystart/SConstruct.

`Fetch(data_file,dir[,ftp_server_info])` A rule to download <`data_file`> from a specific directory <`dir`> of an FTP server <`ftp_server_info`>.
`Flow(target[s],source[s],command[s][,stdin][,stdout])` A rule to generate <`target[s]`> from <`source[s]`> using `command[s]`
`Plot(intermediate_plot[,source],plot_command)` or `Plot(intermediate_plot,intermediate_plots,combination)` A rule to generate <`intermediate_plot`> in the working directory.
`Result(plot[,source],plot_command)` or `Result(plot,intermediate_plots,combination)` A rule to generate a final <`plot`> in the special `Fig` folder of the working directory.
`End()` A rule to collect default targets.

Table 1: Basic methods of an `rsf.proj` object.

```
# Download the input data file
Fetch('lena.img','imgs')

# Create RSF header
Flow('lena.hdr','lena.img',
     'echo n1=512 n2=513 in=$SOURCE data_format=native_uchar',
     stdin=0)

# Convert to floating point and window out first trace
Flow('lena','lena.hdr','dd type=float | window f2=1')

# Display
Result('lena',
       '''
       sfgrey title="Hello, World!" transp=n color=b bias=128
       clip=100 screenratio=1
       ''')

# Wrap up
End()
```

This is our "hello world" example that illustrates the basic use of some of the commands presented in Table 1. The plan for this experiment is simply to download data from a public data server, to convert it to an appropriate file format and to generate a figure for publication. But let us have a closer look at the `SConstruct` script and try to decorticate it.

```
1 from rsf.proj import *
```

is a standard Python command that loads the Madagascar project management module `rsf.proj.py` which provides our extension to SCons.

```
4 Fetch('lena.img','imgs')
```

instructs SCons to connect to a public data server (the default server if no FTP server information is provided) and to fetch the data file `lena.img` from the `data/imgs` directory. Try running "`scons lena.img`" on the command line. The successful output should look like

```
bash$ scons lena.img
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
retrieve(["lena.img"], [])
scons: done building targets.
```

with the target file `lena.img` appearing in your directory. In the following examples, we will use `-Q` (quiet) option of `scons` to suppress the verbose output.

```
7 Flow('lena.hdr','lena.img',
8      'echo n1=512 n2=513 in=$SOURCE data_format=native_uchar',
9      stdin=0)
```

prepares the Madagascar header file `lena.hdr` using the standard Unix command `echo`.

```
bash$ scons -Q lena.hdr
echo n1=512 n2=513 in=lena.img data_format=native_uchar > lena.hdr
```

Since `echo` does not take a standard input, stdin is set to 0 in the Flow command otherwise the first source is the standard input. Likewise, the first target is the standard output unless otherwise specified. Note that `lena.img` is referred as `$SOURCE` in the command. This allows us to change the name of the source file without changing the command.

The data format of the `lena.img` image file is `uchar` (unsigned character), the image consists of 513 traces with 512 samples per trace. Our next step is to convert the image representation to floating point numbers and to window out the first trace so that the final image is a 512 by 512 square. The two transformations are conveniently combined into one with the help of a Unix pipe.

```
12 Flow('lena','lena.hdr','dd type=float | window f2=1')
```

```
bash$ scons -Q lena
scons: *** Do not know how to make target `lena'.  Stop.
```

What happened? In the absence of the file suffix, the `Flow` command assumes that the target file suffix is "`.rsf`". Let us try again.

```
scons -Q lena.rsf
< lena.hdr /RSF/bin/sfdd type=float | /RSF/bin/sfwindow f2=1 > lena.rsf
```

Notice that Madagascar modules `sfdd` and `sfwindow` get substituted for the corresponding short names in the `SConstruct` file. The file `lena.rsf` is in a regularly sampled format[4] and can be examined, for example, with `sfin lena.rsf`[5].

```
bash$ sfin lena.rsf
lena.rsf:
    in="/datapath/lena.rsf@"
    esize=4 type=float form=native
    n1=512          d1=1            o1=0
    n2=512          d2=1            o2=1
        262144 elements 1048576 bytes
```

In the last step, we will create a plot file for displaying the image on the screen and for including it in the publication.

```
15 Result('lena',
16         '''
17         sfgrey title="Hello, World!" transp=n color=b bias=128
18         clip=100 screenratio=1
19         ''')
```

Notice that we broke the long command string into multiple lines by using Python's triple quote syntax. All the extra white space will be ignored when the multiple line string gets translated into the command line. The `Result` command has special targets associated with it. Try, for example, "`scons lena.view`" to observe the figure `Fig/lena.vpl` generated in a specially created `Fig` directory and displayed on the

Figure 2: The output of the first numerical experiment.

screen. The output should look like Figure 2. The reproducible script ends with

```
22 End()
```

Ready to experiment? Try some of the following:

1. Run `scons -c`. The `-c` (clean) option tells SCons to remove all default targets (the `Fig/lena.vpl` image file in our case) and also all intermediate targets that it generated.

   ```
   bash$ scons -c -Q
   Removed lena.img
   Removed lena.hdr
   Removed lena.rsf
   Removed /datapath/lena.rsf@
   Removed Fig/lena.vpl
   ```

   Run `scons` again, and the default target will be regenerated.

   ```
   bash$ scons -Q
   retrieve(["lena.img"], [])
   echo n1=512 n2=513 in=lena.img data_format=native_uchar > lena.hdr
   < lena.hdr /RSF/bin/sfdd type=float | /RSF/bin/sfwindow f2=1 > lena.rsf
   < lena.rsf /RSF/bin/sfgrey title="Hello, World!" transp=n color=b \
   bias=128 clip=100 screenratio=1 > Fig/lena.vpl
   ```

2. Edit your `SConstruct` file and change some of the plotting parameters. For example, change the value of `clip` from `clip=100` to `clip=50`. Run `scons` again and observe that only the last part of the processing flow (precisely, the part affected by the parameter change) is being run:

[4]See `http://rsf.sourceforge.net/wiki/index.php/Format`

[5]See `http://rsf.sourceforge.net/wiki/index.php/Programs#sfin`.

```
bash$ scons -Q view
< lena.rsf /RSF/bin/sfgrey title="Hello, World!" transp=n color=b \
bias=128 clip=50 screenratio=1 > Fig/lena.vpl
/RSF/bin/xtpen Fig/lena.vpl
```

SCons is smart enough to recognize that your editing did not affect any of the previous results in the data flow chain! Keeping track of dependencies is the main feature that separates data processing and computational experimenting with SCons from using linear shell scripts. For computationally demanding data processing, this feature can save you a lot of time and can make your experiments more interactive and enjoyable.

3. A special parameter to SCons (defined in `rsf.proj.py`) can time the execution of each step in the processing flow. Try running `scons TIMER=y`.

4. The `rsf.proj` module has direct access to the database that stores parameters of all Madagascar modules. Try running `scons CHECKPAR=y` to see parameter checking enforced before computations[6].

The summary of our SCons commands is given in Table 2.

Example 2

The plan for this experiment is to add random noise to the test "Lena" image and then to attempt removing it by low-pass filtering and by hard thresholding of coefficients in the Fourier domain. The result images are shown in Figure 3 and 4.

Since the `SConstruct|` file is a Python script, we can also use all the flexibility and power of the Python language in our Madagascar reproducible scripts. A demo script is available in the `rsf/scons/rsfpy` subdirectory of the Madagascar `book` directory. Rather than commenting it line-by-line, we select some parts of interest.

In the `SConstruct` script, we can declare Python variables

```
3 bias = 128
```

and use them later, for example, to define our customized plot command as a Python function

```
5  def grey(title,transp='n',bias=bias):
6      return '''
7      sfgrey title="%s" transp=%s bias=%g clip=100
8      screenht=10 screenwd=10 crowd2=0.85 crowd1=0.8
9      label1= label2=
10     ''' % (title,transp,bias)
```

[6]This feature is new and experimental and may not work properly yet

`scons <file>` Generate `<file>` (usually requires `.rsf` suffix for `Flow` targets and `.vpl` suffix for `Plot` targets.)
`scons` Generate default targets (usually figures specified in `Result`.)
`scons view` or `scons <result>.view` Generate `Result` figures and display them on the screen.
`scons print` or `scons <result>.print` Generate `Result` figures and print them.
`scons lock` or `scons <result>.lock` Generate `Result` figures and install them in a separate location.
`scons test` or `scons <result>.test` Generate `Result` figures and compare them with the corresponding "locked" figures stored in a separate location (regression testing).
`scons <result>.flip` Generate the `<result>` figure and compare it with the corresponding "locked" figure stored in a separate location by flipping between the two figures on the screen.
`scons TIMER=y ...` Time the execution of each step in the processing flow (using the Unix `time` utility.)
`scons CHECKPAR=y ...` Check the names and values of all parameters supplied to Madagascar modules in the processing flows before executing anything (guards against incorrect input.) This option is new and experimental.

Table 2: SCons commands and options defined in `rsf.proj`.

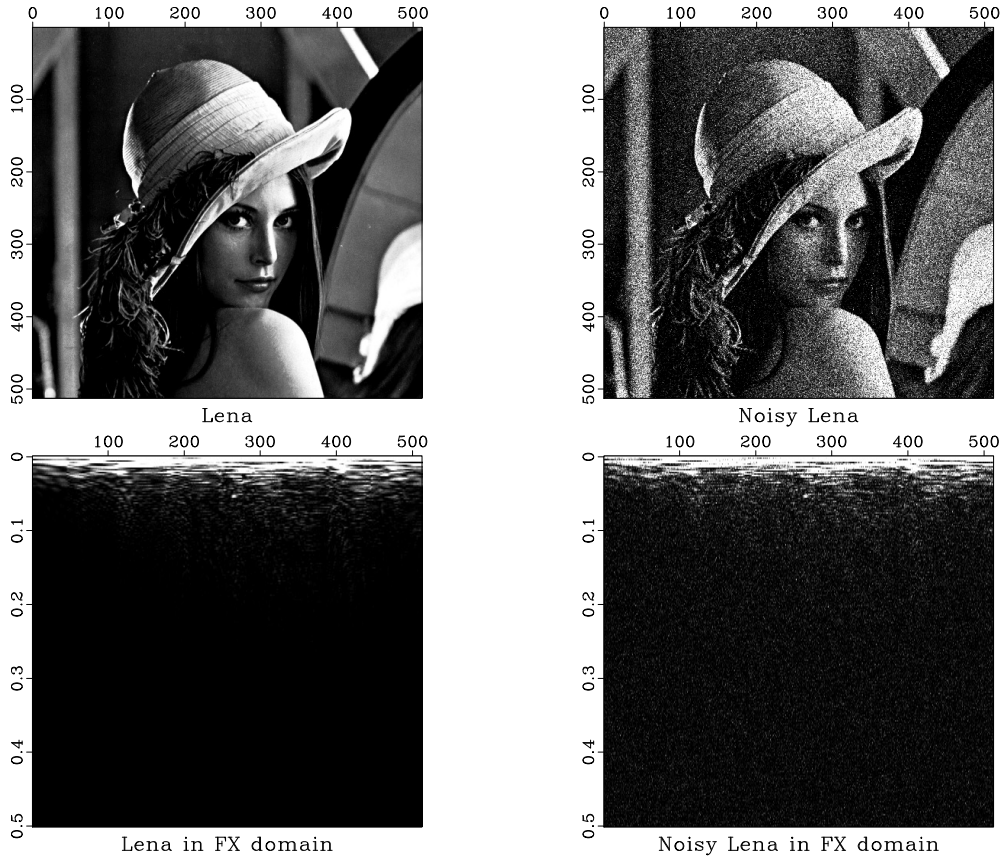

Figure 3: Top left: original image. Top right: random noise added. Bottom left: original image spectrum in the Fourier (F-X) domain. Bottom right: noisy image spectrum in the Fourier (F-X) domain.

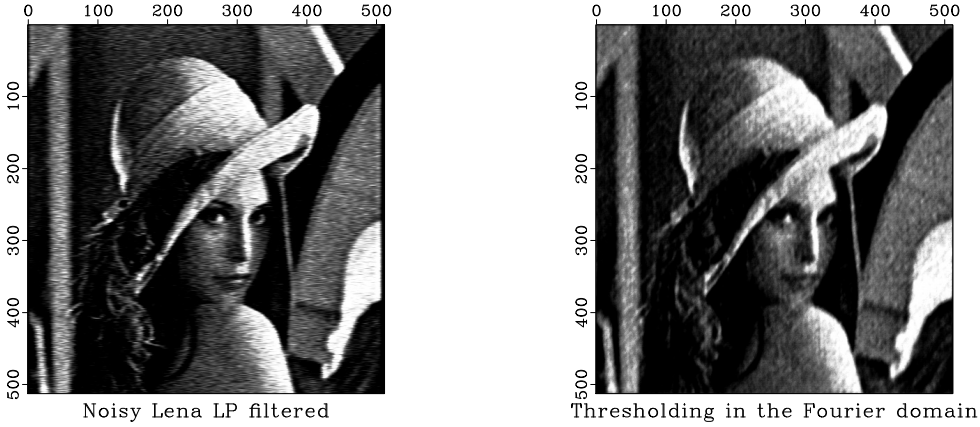

Figure 4: Left: denoising by low-pass filtering. Right: denoising by hard thresholding in the Fourier domain.

This Python function, named `grey()`, can then be called in Plot or Result commands, e.g.

```
48 Plot('lplena',grey('Noisy Lena LP filtered'))
```

We can define a Python dictionary, e.g.

```
34 titles = {'lena':'Lena',
35            'nlena':'Noisy Lena'}
```

and loop over its entries, e.g.

```
36 for name in titles.keys():
37     Plot(name,grey(titles[name]) )
38     cftitle = titles[name]+' in FX domain'
39     Flow('fx'+name,name,'sfspectra')
40     Plot('fx'+name,grey(cftitle ,'y',100))
```

Note that the title of the plots is obtained by concatenating Python strings.

Python strings can also be used to define sequences of commands used in several Flows, e.g.

```
65 # 2-D FFT
66 fft2 = 'sffft1 sym=y | sffft3 sym=y'
67 Flow('fnlena','nlena',fft2)
```

Finally, in our Madagascar reproducible script, we may want the option to pass command line arguments when running SCons or use default values otherwise, e.g.

```
69 # denoising using thresholding in the Fourier domain
70 fthr = float(ARGUMENTS.get('fthr', 70))
71 Flow('fthrlena','fnlena','sfthr thr=%f mode="hard"' % fthr)
```

Running `scons` only, the default value set for fthr (i.e. 70) is used whereas running `scons fthr=68` set fthr to a command line specified value.

This is by no mean an exhaustive list of options but, hopefully, it gives you a flavor of the powerful tool you have in hands. Enjoy!

CREATING REPRODUCIBLE DOCUMENTATION

You are done with computational experiments and want to communicate them in a paper. SCons helps us create high-quality papers, where computational results (figures) are integrated with papers written in LaTeX. The corresponding SCons extension is defined in `$RSFROOT/lib/rsftex.py` where `RSFROOT` is the environmental variable to the Madagascar installation directory. The source of this file is in python/rsftex.py. We summarize the basic methods and commands in Tables 3 and 4.

`Paper(<paper_name>,[,lclass][,use][,include][,options])` A rule to compile `<paper_name>.tex` LaTeX document using the LaTeX2e class specified in `lclass` (default is `geophysics.cls` from the SEGTeX package) with additional options specified in `options`, additional packages specified in `use`, and additional preamble specified in `include`
`End()` A rule to collect default targets (referring to `paper.tex` document).

Table 3: Basic methods of an `rsf.tex` object.

`scons` Generate the default target (usually the PDF file `paper.pdf` from the source LaTeX file `paper.tex`.)
`scons pdf` or `scons <paper_name>.pdf` Generate PDF files from LaTeX sources `paper.tex` or `<paper_name>.tex`.
`scons read` or `scons <paper_name>.read` Generate PDF files from LaTeX sources `paper.tex` or `<paper_name>.tex` and display them on the screen.
`scons print` or `scons <paper_name>.print` Generate PDF files from LaTeX sources `paper.tex` or `<paper_name>.tex` and print them.
`scons html` or `scons <paper_name>.html` Generate HTML files from LaTeX sources `paper.tex` or `<paper_name>.tex` using LaTeXtoHTML. The directory `<paper_name>_html` gets created.
`scons install` or `scons <paper_name>.install` Generate PDF and HTML files from LaTeX sources `paper.tex` or `<paper_name>.tex` and install them in a separate location (used for publishing on a web site).
`scons wiki` or `scons <paper_name>.wiki` Convert LaTeX sources `paper.tex` or `<paper_name>.tex` to the `MediaWiki` format (used for publishing on a Wiki web site).

Table 4: SCons commands defined in `rsf.tex`.

Example

This paper by itself is an example of a reproducible document. It is generated using the following `SConstruct` file which is place in the directory above the projects directories.

```
from rsf.tex import *
Paper('velan',use='hyperref,listings,color')
End(use='hyperref,listings,color',
    color='modl modl2 cdp1500 cdp2000 cdp2500 cdp3000 cdp3500 pick')
```

This `SConstruct` generates this paper but it can also compile `velan.tex` in the same directory. Note that there is no `Paper` command for `paper.tex` since it is the default documentation name. Optional LaTeX packages and style used in `paper.tex` are passed in the End command.

Let's now have a closer look at `paper.tex` to understand how the figures of the documentation are linked to the reproducible scripts that created them. First of all, note that `paper.tex` is not a regular LaTeX document but only its body (no \documentclass, \usepackage, etc.). In our paper, Fig. 2 was created in the project folder `easystart` (sub-folder of our documentation folder) by the result plot `lena.vpl`. In the LaTeX source code, it translates as

```
\inputdir{easystart}
\sideplot{lena}{height=.25\textheight}{The output of the first
numerical experiment.}
```

The \inputdir command points to the project directory and the \sideplot command calls <`result_name`>. The LaTeX tag of the figure is `fig:`<`result_name`>. The first time the paper is compiled, the result file is automatically converted to the PDF file format.

REFERENCES

Bivand, R., 2006, Implementing spatial data analysis software tools in r: Geographical Analysis, **38**, 23–40.

Buckheit, J., and D. L. Donoho, 1995, Wavelab and reproducible research, *in* Wavelets and Statistics: Springer-Verlag, **103**, 55–81.

Claerbout, J., 1992a, Electronic documents give reproducible research a new meaning: 62nd Ann. Internat. Mtg, Soc. of Expl. Geophys., 601–604.

———, 1992b, How to use Cake with interactive documents, *in* SEP-73: Stanford Exploration Project, 451–460.

Claerbout, J. F., and M. Karrenbach, 1993, How to use cake with interactive documents, *in* SEP-77: Stanford Exploration Project, 427–444.

Claerbout, J. F., and D. Nichols, 1990, Why active documents need cake, *in* SEP-67: Stanford Exploration Project, 145–148.

Dubois, P. F., 2003, Why Johnny can't build: Computing in Science & Engineering, **5**, 83–88.

Fomel, S., M. Schwab, and J. Schroeder, 1997, Empowering SEP's documents, *in* SEP-94: Stanford Exploration Project, 339–361.

Gentleman, R. C., V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang, 2004, Bioconductor: open software development for computational biology and bioinformatics: Genome Biology, **5**, R80.

Hubbard, B. B., 1998, The world according to wavelets: The story of a mathematical technique in the making: AK Peters.

Knuth, D. E., 1984, Literate programming: Computer Journal, **27**, 97–111.

LeVeque, R. J., to appear, 2006, Wave propagation software, computational science, and reproducible research: Presented at the Proc. International Congress of Mathematicians.

Mallat, S., 1999, A wavelet tour of signal processing: Academic Press.

Nichols, D., and S. Cole, 1989, Device independent software installation with CAKE, *in* SEP-61: Stanford Exploration Project, 341–344.

Raymond, E. S., 2004, The art of UNIX programming: Addison-Wesley.

Rossum, G. V., 2000a, Python reference manual: Iuniverse Inc.

———, 2000b, Python tutorial: Iuniverse Inc.

Scales, J. A., and H. Ecke, 2002, What programming languages should we teach our undergraduates?: The Leading Edge, **21**, 260–267.

Schwab, M., M. Karrenbach, and J. Claerbout, 2000, Making scientific computations reproducible: Computing in Science & Engineering, **2**, 61–67.

Schwab, M., and J. Schroeder, 1995, Reproducible research documents using GNUmake, *in* SEP-89: Stanford Exploration Project, 217–226.

Sigmon, K., and T. A. Davis, 2001, MATLAB primer, sixth edition: Chapman & Hall.

Stallman, R. M., R. McGrath, and P. D. Smith, 2004, GNU make: A program for directing recompilation: GNU Press.

Thimbleby, H., 2003, Explaining code for publication: Software - Practice & Experience, **33**, 975–908.

van der Linden, P., 1994, Expert C programming: Prentice Hall.