Making scientific computations reproducible

Matthias Schwab, Martin Karrenbach, and Jon Claerbout [1]

ABSTRACT

To organize computational scientific research and hence to conveniently transfer our technology, we impose a simple filing discipline on the authors in our laboratory. A document's makefile includes laboratory-wide standard rules that offer readers these four standard commands: `make burn` removes the document's result figures, `make build` recomputes them, `make view` displays the figures, and `make clean` removes any intermediate files. Although we developed these standards to aid readers we discovered that authors are often the principal beneficiaries.

INTRODUCTION

In the mid 1980's, we noticed that a few months after completing a project, the researchers at our laboratory were usually unable to reproduce their own computational work without considerable agony. In 1991, we solved this problem by developing a concept of electronic documents that makes scientific computations reproducible. Since then, electronic reproducible documents have become our principal means of technology transfer of scientific computational research. A small set of standard commands makes a document's results and their reproduction readily accessible to any reader. To implement reproducible computational research the author must use makefiles, adhere to a community's naming conventions, and reuse (include) the community's common building and cleaning rules. Since electronic reproducible documents are reservoirs of easily maintained, reusable software, not only the reader but also the author benefits from reproducible documents.

REPRODUCIBLE RESEARCH

On average, two PhD students graduate each year from our laboratory, the *Stanford Exploration Project*. Years ago, junior students who built on their seniors' work

[1] **email:** matt@sep.Stanford.EDU, martin@sep.Stanford.EDU, jon@sep.Stanford.EDU

often spent a considerable effort to merely reproduce their colleagues' old computational results.

Indeed, the problem occurs wherever traditional methods of scientific publication are used to describe computational research. In a traditional article the author merely outlines the relevant computations: the limitations of a paper medium prohibit a complete documentation including experimental data, parameter values, and the author's programs. Consequently, the reader has painfully to re-implement the author's work before verifying and utilizing it. Even if the reader receives the author's source files (a feasible assumption considering the recent progress in electronic publishing), the results can be recomputed only if the various programs are invoked exactly as in the original publication. The reader must spend valuable time merely rediscovering minutiae, which the author was unable to communicate conveniently.

To facilitate efficient technology transfer, our laboratory developed the concept of a *reproducible electronic document (ReDoc).* A reader of a reproducible document can remove and rebuild the document's results without any application-specific knowledge. An author whose research involves scientific computations on a UNIX computer can easily create reproducible documents. Beyond a traditional article and the application's source code, a reproducible document contains three additional components: (1) makefiles, (2) a small set of universal `make` rules (less than 100 lines), and (3) naming conventions for files.

A makefile contains the commands to build a software package and is a standard UNIX utility for software maintenance. More powerful than a simple script of commands, a makefile has a notion that result files (*targets*) are up to date when they are younger than their corresponding source files (*dependencies*). Since maintaining a reproducible research project resembles maintaining software, we find the `make` utility crucial to solving our problem elegantly. Fortunately, fine tutorial books about the `make` language exist (Stallman and McGrath, 1991; Oram and Talbott, 1991), and students easily begin using `make` even without formal introduction.

The second component in making scientific computations reproducible is the consistent availability of a small set of standard `make` rules. These rules allow a reader to interact with the document without knowing the underlying application-specific commands or files. The inclusion of an universal set of rules in every makefile ensures the rules' consistency across documents and enables authors to concentrate exclusively on the application-specific part of the makefile. Furthermore, a central set of rules accumulates the wisdom of a community on how to organize a reproducible electronic document. Our laboratory offers about 100 lines of GNU `make` code (*ReDoc rules*) that, when included in the application makefile, implement a simple but powerful reader interface. In the electronic version of any reproducible electronic document, these ReDoc rules facilitate four commands: `make burn` removes the result files (usually figures), `make build` recomputes them, `make view` displays the figures, and `make clean` removes any files that are neither source nor result files. The process of recomputing the author's results allows a reader to understand and to modify the interaction of the various components.

The third component of a reproducible document system is its naming conventions for all files. These conventions allow a community to formulate universal rules that recognize a file's type and handle it appropriately. For example, a cleaning rule removes all intermediate files. The rule identifies intermediate files based on a community's naming conventions: typically files with suffix `.o` (object files) or files with the name stem `junk` (temporary files) are removed. A cleaning rule is important since a cleaned directory is more accessible and inviting to a reader than a cluttered uncleaned one. Furthermore, a cleaning rule saves resources such as disk memory by removing superfluous files. Naming conventions are also needed for result files. The rules for displaying, removing, or recomputing a result file are based on the result's file name. For example, at our laboratory result files (which are invariably figures) can have various formats such as postscript or gif. Our laboratory's naming conventions require the author to indicate the result file's format by a suffix, such as `.ps` or `.gif`. Consequently our laboratory can supply a universal format-independent rule for displaying result files: The rule identifies the result file's suffix, concludes the file's format, and invokes the appropriate viewing program such as `ghostview` for postscript or `xview` for gif files.

ReDoc rules are easy to use. An author who already uses makefiles only needs to adhere to the ReDoc naming conventions and include the ReDoc rules to make a traditional document reproducible. Our laboratory distributes its ReDoc rules, this article, and the accompanying example on its World Wide Web site (Schwab and Claerbout, 1996) (see Figure 1). A different community may need to adapt these ReDoc rules to its own peculiarities and naming conventions.

At our laboratory the software readily accessible to any researcher has increased tremendously. Today students commonly take up projects of former students, starting by easily removing and recomputing the original result files. Students who graduated and left our laboratory were able to seamlessly continue their own research at their new locations.

We successfully employed the ReDoc rules in our laboratory's most recent sponsor report (14 articles by 15 authors) and three of Jon Claerbout's textbooks on seismic imaging. These documents contain a total of 483 result files: 276 easily reproducible, 21 conditionally reproducible, and 186 non-reproducible figures. Before publication automatic scripts removed and rebuilt all 276 easily reproducible result files (see Figure 2). We use the same scripts and documents to benchmark computer platforms. Additionally, our laboratory published 12 PhD theses that use an earlier version of the reader interface based on an dialect of `make` called `cake`. These electronic documents are available on CD-ROMs (Claerbout, 1996).

We chose to implement the current reader interface in GNU `make`, since it is platform-independent, excels in the efficient maintenance of even complex software packages, and is equipped with a special mechanism to handle intermediate files, which we will discuss later (see section *Simultaneously clean and up to date*). Conceptually the ReDoc reader interface is independent of the document format (TEX, html, etc.) and independent of the underlying computational software, such as Matlab, Mathematica, or C and FORTRAN programs. Even though this paper restricts itself to UNIX systems and the `make` utility, the concept of a reader interface to reproduce a document's computational results should apply to electronic documents in other computer environments as well.

What is next? Of course, we want to publish our results on the World Wide Web. The Web conveniently distributes the combination of reading material for researchers and software for computers. Ideally, each computed figure in a future World Wide Web document should be accompanied by a *push-button* for the `burn, build, clean`, and `view` command. Currently we are closely watching the development of Java (SUN, 1996), a computer language for software on the Internet.

BENEFITS TO THE READER

Just as a driver wants to find the brake pedal at the same location in every car, a reader of research documents wants a few standard commands to explore the scientific contents of any electronic document. The ReDoc rules offer four such standard commands:

- `make burn` removes the result files such as the postscript and gif version of a computed figure.
- `make build` recomputes the result files. It compiles the necessary programs, correctly invokes a sequence of executables, and finally stores the results in predefined files.
- `make view` updates the result files, if necessary, and displays them using the appropriate viewer, such as `ghostview` for a postscript figure, or the standard `cat` UNIX program for an ASCII file.
- `make clean` removes all the intermediate files that were created during the execution of the `build` command. It leaves the result and source files intact.

Consistent standard commands to remove and reproduce a document's result files not only help a reader access and study an unknown document, but also enable an author maintain his own software. A reproducible document is a research and software filing system. Authors document their scientific computations in the article and preserve the computational details in fully functional examples. The standardized commands of the ReDoc reader interface allow authors to easily test their archived research software by occasionally removing and regenerating the document's results. A community can even develop automatic scripts to verify any document's completeness and reproducibility before its publication (see Figure 2). Publishers may envision that an electronic scientific journal could be refereed by testing the reproducibility of its illustrations.

Simultaneously clean and up to date

A document is best maintained both clean and up to date: each result file is younger than its ultimate source files, and the intermediate files are removed, but assumed to be up to date. A reader prefers a clean directory, since an uncluttered directory clearly presents the important source and result files. Simultaneously a reader expects reassurance that the document's results correspond with the existing source files. (Furthermore, we found that only cleaned documents are functional on a CD-ROM: intermediate files once stored on the read-only memory of a CD-ROM cannot be overwritten when the files are later regenerated by a reader.)

Unfortunately, popular `make` dialects generally do not support rules to keep documents simultaneously clean and up to date. These dialects consider a result file out of date when certain intermediate files are missing (for example, GNU `make` considers a result file out of date if a file is absent whose dependency is formulated by a non-pattern rule). Such treatment of intermediate files is convenient for software maintenance but not suitable for reproducible electronic documents.

The `cake` dialect of `make` was designed for document maintenance rather than for software maintenance. `cake` assumes *all* absent intermediate files to be up to date and therefore supports documents that are simultaneously clean and up to date. `cake` was originally introduced at our laboratory because it was the first freely distributed, platform-independent `make` dialect we found. Unfortunately, `cake`'s limited popularity at other sites made our reproducible electronic documents unattractive to potential readers.

Today's ReDoc rules are able to maintain a document clean and up to date while being formulated in the very popular GNU `make` dialect. At our request, Richard Stallman recently enhanced GNU `make` to adequately handle the ReDoc rules' intermediate targets.[2] He added a special built-in target, `.SECONDARY` that allows the author to choose the behavior of GNU `make` with respect to its missing intermediate files. If a makefile includes a `.SECONDARY` target without dependencies (the default at our laboratory), then every missing intermediate file is presumed up to date. The `.SECONDARY` target is implemented in GNU `make` versions higher than 3.74. The GNU `make` version we use is available on our World Wide Web site (Schwab and Claerbout, 1996).

Degree of reproducibility

Since our laboratory deals with computational problems of various sizes using a diverse collection of software and hardware tools, not all result files are easily reproducible for every reader. Consequently, application makefiles typically define three result list variables: `RESULTSER`, `RESULTSCR`, and `RESULTSNR`. The endings `ER`, `CR`, and `NR` indicate to the reader the degree of reproducibility:

[2] GNU `make` refers to intermediate files as *secondary files*. It uses the word *intermediate* in a slightly different, more restrictive sense.

- **ER: Easily reproducible** result files can be regenerated within ten minutes on a standard workstation.
- **NR: Non-reproducible** result files, such as hand-drawn illustrations or scanned figures, cannot be recalculated by the reader.
- **CR: Conditionally reproducible** result files require proprietary data, licensed software, or more than 10 minutes for their recomputation. The author nevertheless supplies a complete set of source files and rules to ensure that a reader can reproduce the results if he possesses the necessary resources. Additionally the author lists these resources in a warning file (e.g. `myresult.warning`), which accompanies the conditionally reproducible result file (e.g., `myresult.ps`). In industrial-scale geophysical research, many interesting results are conditionally reproducible.

The standard make targets `burn` and `build` are complemented by targets `burnER, burnCR, burnNR, burnall` and `buildER, buildCR, buildNR, buildall`. For example `burnCR` burns all conditionally reproducible result files. The target `burn` is defaulted to `burnER` to restrict the standard removal of result files to easily reproducible ones. The target `build` is defaulted to `buildER` to recompute the result files that `make burn` removes.

EFFORT REQUIRED BY THE AUTHOR: AN EXAMPLE

The reader interface demands minimal effort from the author. The author merely supplies the application-specific rules, while the community's ReDoc rules contain all definitions that are not application-specific. The author's definitions have to conform to the community's naming conventions, so that the author's application-specific rules can be invoked by the universal ReDoc rules.

The author of a reproducible document has to list each result file as either easily, conditionally, or non-reproducible. For every easily or conditionally reproducible result, the author has to supply a rule that generates the result file. Furthermore, the author needs to specify a cleaning rule that removes the intermediate files. The ReDoc rules offer the author a comprehensive default cleaning rule, `jclean`.

Since the author's rules deal with the document's application, the effort required of an author is best illustrated by an example. The electronic version of this article is accompanied by a subdirectory called *Frog*. Frog contains a complete albeit small reproducible electronic document about a finite-difference approximation of the 2-D surface waves caused by a frog hopping around a rectangular pond. The files `paper.latex` and `paper.ps` hold two formats of the short scientific article describing the finite-difference approximation (see Inset). Some RATFOR[3] files implement

[3]RATFOR is a preprocessor for FORTRAN that provides control flow constructs similar to C. Many UNIX systems have the original AT&T RATFOR. Our laboratory distributes a freely available RATFOR on its World Wide Web server.

the 2-D wave propagation code. The `Fig` directory contains the result files: a figure (postscript and gif version) of the pond after some wild hops by the frog and the output (two float numbers) of a dot-product test of the linear finite-difference operator and its adjoint. To organize the document's files, the author of the Frog example wrote the following makefile:

```
SEPINC = ../Rules
include ${SEPINC}/Doc.defs.top

RESULTSER = frog dot

col = 0.,0.,0.-1.,1.,1.
${RESDIR}/frog.ps ${RESDIR}/frog.gif: frog.x
        frog.x                    > junk.pgm
        pgmtoppm ${col} junk.pgm > junk.ppm
        ppmtogif        junk.ppm > ${RESDIR}/frog.gif
        pnmtops         junk.pgm > ${RESDIR}/frog.ps

objs =  copy.o adjnull.o rand01.o wavecat.o \
        pressure.o velocity.o viscosity.o wavesteps.o
frog.x: ${objs}

dot.build ${RESDIR}/dot.txt : dot.x
        dot.x dummy > ${RESDIR}/dot.txt
dot.view: ${RESDIR}/dot.txt
        cat ${RESDIR}/dot.txt
dot.burn:
        rm ${RESDIR}/dot.txt

dot.x : ${objs}

clean: jclean

include ${SEPINC}/Doc.rules.red
include ${SEPINC}/Doc.rules.idoc
include ${SEPINC}/Prg.rules.std
```

The variable `RESULTSER` contains the list of the document's easily reproducible results, `frog` and `dot`.

The next rule contains the commands to build the postscript and gif version of the `frog` result. Such a rule is application-specific and cannot be supplied by included default rules. The target names comprise the directory `RESDIR`, in which the result files reside, and file suffixes (`.ps, .gif`), which indicate the files' formats. The rule depends on an executable `frog.x`, which it executes during the computations of the result.

Default rules for compilation and linking of executables such as `frog.x` are supplied by a shared include file, `Prg.rules.std`[4]. The dependency of the executable on its subroutine object files, as in the case of `frog.x`, needs to be defined by the author of the makefile, since it depends on the application-specific file names.

In the case of the `dot` result file, the rules supplied by the author reflect the commands of the reader interface: `dot.build` creates the result file, `dot.view` displays it, and `dot.burn` removes it.

[4]Compilation and link rules are compiler dependent. In the Frog example, we include some generic FORTRAN rules; at our laboratory compilation rules depend on an environment variable indicating the compiler type.

Finally, the target `clean` invokes the included default target `jclean`. The targets remove intermediate files based on our laboratory's naming conventions.

OUR LABORATORY'S REDOC RULES

In the Frog example, a reader invokes targets, such as `build`, that are not listed in the author's application makefile. These targets are supplied by our laboratory's ReDoc rules and are merely included in the document's makefile (`Doc.defs.top, Doc.rules.red, Doc.rules.idoc`). They ensure a consistent reader interface, prevent the author from re-implementing the ReDoc rules in every makefile, and accumulate the wisdom of the entire community. They are formulated in a way that any individual author can override them. Our experience shows, however, that overriding is hardly ever necessary or desirable.

An author or reader does not need to know the implementation details of the ReDoc rules to use the rules (most researchers at our laboratory have never inspected the ReDoc rules). But you may wonder how the rules operate and how you may have to adapt the rules for your community's computational environment.

Burn

`make burn` invokes a chain of rules, which ultimately finds all easily reproducible result files and removes them. The `burn` target is included in every application makefile as part of the ReDoc rules.

```
burn: burnER

burnER: ${addsuffix .burn, ${RESULTSER}}
burnCR: ${addsuffix .burn, ${RESULTSCR}}

%.burn:
        ${foreach sfx, ${RES_SUFFIXES} , \
          if ${EXIST} ${RESDIR}/$*${sfx} ; then \
             ${RM}    ${RESDIR}/$*${sfx} ;  fi; \
          }
```

The `burn` target invokes its dependency `burnER`. The `burnER` rule selects the easily reproducible result files for removal. The `burnER` rule uses GNU `make`'s built-in function `addsuffix` to generate its dependency list. Each entry of `burnER`'s dependency list is a concatenation of the name of an easily reproducible file and the suffix `.burn`. In the Frog example, `burnER` depends on `frog.burn` and `dot.burn`. The dependency `frog.burn` invokes the pattern rule `%.burn`, which removes the result files corresponding to the result `frog`. At our laboratory a single result name such as `frog` usually denotes several result files of identical contents but differing format, e.g. postscript or gif. The `%.burn` rule scans a list of possible suffixes (`RES_SUFFIXES = .ps .gif`) and removes all related result files: `frog.ps` and `frog.gif`.

Since text result files, such as `dot.txt`, are rare at our laboratory, the ReDoc rules do not contain laboratory-wide rules for handling them. Consequently the author of the Frog document supplies an explicit `dot.burn` rule in the makefile. This explicit `dot.burn` rule overrides the default `%.burn` pattern rule, which generates postscript result files.

The standard `burn` rule exclusively removes the easily reproducible result files. A reader can remove any existing, conditionally reproducible result files by invoking `make burnCR`. A reader can exclusively remove the result files related to the `frog` result by invoking `make frog.burn`.

Build

The `build` rule updates, if necessary, the document's easily reproducible result files. The implementation of the `build` rule is similar to the implementation of the `burn` rule:

```
build:   buildER
buildER: ${addsuffix .build, ${RESULTSER}}
buildCR: ${addsuffix .build, ${RESULTSCR}}

%.build: ${RESDIR}/%.ps
```

At our laboratory, almost every result file is a figure that exists in postscript format. Consequently the ReDoc `%.build` rule updates the postscript version of any easily reproducible result, such as `${RESDIR}/frog.ps`. Additional versions of the result (e.g. `frog.gif`) are usually generated as a side effect of the rule that computes the postscript version (`frog.ps`).

As in the case of `dot.burn`, the nonstandard text result `dot` (since it is not a figure) requires the author to supply an explicit `dot.build` rule.

View

The `view` rule updates and displays the results:

```
view  : ${addsuffix  .view, ${RESULTSALL}}

%.view: FORCE
        if   ${CANDO_GIF}    ; then \
          ${MAKE} $*.viewgif ; \
        elif ${CANDO_PS}     ; then \
          ${MAKE} $*.viewps  ; \
        else \
          echo "can't make $*.viewps $*viewgif";\
        fi
```

`RESULTSALL` lists all result files and is defined as the concatenation of `RESULTSNR`, `RESULTSCR`, and `RESULTSER`.

At our laboratory the `%.view` rule checks for the various formats of a result and chooses the first version the makefile knows how to generate. The variable `CANDO` contains the return value of a recursive `gmake -n` call. This return value indicates if that particular version of the result can be built. Having found a version that can be built, the `%.view` rule invokes another rule (`%.viewgif` or `%.viewps`) that updates and displays the result file:

```
%.viewgif  : ${RESDIR}/%.gif FORCE
        ${XVIEW} ${UXVIEWFLAGS} ${RESDIR}/$*.gif

%.viewps : ${RESDIR}/%.ps FORCE
        ${GVIEW} ${UGVIEWFLAGS} ${RESDIR}/$*.ps
```

In the Frog example, the `frog.view` rule finds a rule for computing a `.gif` version of `frog`. Consequently, it invokes the gif rule `frog.viewgif`. In return `frog.viewgif` executes `xview` to display the result file `frog.gif`. If your computer system does not support the gif viewer `xview`, then you will need to supplement the `%.viewgif` rule with your own display command. Alternatively, `frog.viewps` executes `ghostview` to display the result file `frog.ps`.

Clean

A community's cleaning rule is designed to remove the intermediate files and thereby to isolate the source and result files. A universal cleaning rule attempts to recognize the intermediate files according to the community's naming conventions. Unfortunately, such a rule cannot possibly anticipate all names the author may choose for his intermediate files. Consequently, our laboratory does not supply a fixed, universal `clean` rule, but a `jclean` (*Jon's clean*) rule. `jclean` removes the files that adhere to our laboratory's naming convention for intermediate files. Every author is responsible to implement his own `clean` rule. Most authors at our laboratory accept the default cleaning rule by defining `clean` as:

```
clean: jclean
```

Some authors at our laboratory append the default `jclean` with a command to remove some additional files that do not adhere to the standard naming conventions. Only very few authors ignore the `jclean` target (and its communal wisdom) and design their own rule.

Since the author of the Frog example adheres strictly to the ReDoc naming conventions for files, the default `jclean` mechanism suffices to remove the intermediate files:

```
jclean : klean.usual klean.fort ;

KLEANUSUAL := core a.out paper.log *.o *.x *.H *.ps *.gif
klean.usual :
        @-${TOUCH} ${KLEANUSUAL} junk.quiet
        @-${RM}    ${KLEANUSUAL} junk.*
```

```
FORT_FILES = $(patsubst %.f,%,$(wildcard *.f)) junk.quiet
klean.fort:
	@\
	for name in ${FORT_FILES} ; do \
	  if  ${EXIST} $${name}.r ; then  \
	      ${TOUCH} $${name}.f ; \
	      ${RM}    $${name}.f ; \
	  fi ; \
	done
```

The `jclean` target uses two methods to identify intermediate files. The first method, `klean.usual`, simply removes files whose names fit one of the rule's name patterns: e.g. the executable `frog.x`, or the intermediate bitmap files `junk.pgm` and `junk.ppm`. The second method, `klean.fort`, removes FORTRAN files, such as `frog.f`, if RATFOR versions of the program, such as `frog.r`, exist.

We are currently collaborating with Richard Stallman of the Free Software Foundation to develop an alternative, more reliable cleaning mechanism. This alternative mechanism would free the author from naming the intermediate files according to the community's naming conventions. The anticipated cleaning mechanism analyses the makefile's rules and dependencies to identify the intermediate files. Fastidious authors would have the option of automatically removing all files that are neither source files nor result files.

ACKNOWLEDGMENTS

We appreciate Richard Stallman's advice and his implementation of the special built-in target `.SECONDARY`. Joel Schroeder conceived the three result lists and understood precedence of GNU `make` definitions. Dave Nichols discovered `cake` and taught our laboratory how to use it. Steve Cole and Dave Nichols wrote xtpanel (Cole and Nichols, 1996) and helped wrap our tools into an interactive, electronic book (see Figure 3).

REFERENCES

Claerbout, J. F., 1996, CDROMs of the Stanford Exploration Project: http://sepwww.stanford.edu/office/sepcd.html/.

Cole, S. P., and Nichols, D., 1996, The xtpanel page: http://sepwww.stanford.edu/oldsep/dave/xtpanel.

Oram, A., and Talbott, S., 1991, Managing projects with make: O'Reilly & Associates, Inc.

Schwab, M., and Claerbout, J. F., 1996, SEP's reproducible electronic documents: http://sepwww.stanford.edu/redoc/.

Stallman, R. M., and McGrath, R., 1991, GNU Make: Free Software Foundation.

SUN, 1996, Java: Programming for the internet: http://java.sun.com/.

Fig 1: The GNU `make` code that implements the ReDoc rules, this article, and its example are freely available on our World Wide Web site: `http:/sepwww.stanford.edu/pub/redoc/`.

Fig 2: A concrete test of a document's reproducibility is a cycle of burning and rebuilding its results. A simple script can implement such a reproducibility test by invoking the ReDoc rules described in this article. The ReDoc rules remove and regenerate the document's results independent of the document's content. The graph above plots the successfully reproduced figures versus the series of tests that removed and rebuilt the figures. The document contained 14 articles with 112 easily reproducible figures by 15 authors. After each test the authors were given time for corrections. After the first test, only 60% of the document's easily reproducible figures were in fact reproducible. After the fourth test, almost all figures were reproducible and the document was published.

Fig 3: The reader interface for reproducible research is only one component of SEP's current computational research environment: A research document at SEP is written in LaTeX(visible in the background to the left). Using SEP's own LaTeX macros, a push-button in each figure caption invokes a graphic user interface (written in a script language called *xtpanel*). The graphic user interface enables a reader to interactively execute the `burn`, `build`, `clean`, and `view` commands for each individual figure. (The panel is shown in the foreground. The result of `make view` is shown towards the right.) SEP's GNU `make` rules allow an author to easily extend the interactivity of a result figure to additional, application-specific actions. Unfortunately these features are beyond the scope of this article. However, we distribute our collection of software and the theses of our research group on CD-ROMs.