Reproducible documents and interactive graphics on the World Wide Web

Matthias Schwab[1]

keywords: *not available*

ABSTRACT

Reproducible documents, interactive graphics, and the Internet enables computational researchers to communicate and share their research easily. A reproducible document offers a reader three basic commands when inspecting any computational result: The first command, `view`, displays the result inviting the reader to scrutinize it. The second command, `build` recomputes the result from a set of given source files, such as experimental data and a processing program. The third command, `info`, offers the reader details on the result, such as the author's name, a time estimate how long a reproduction of the result may take, and possibly an URL which dispenses more information on the overall research project. My Internet version of a reproducible document includes the three reproducibility commands as pushbuttons. The underlying Java applet downloads and executes Java applications that process and display the results. A reader can step by step explore in depth a new technology before downloading and installing the technology's software. An author can maintain her collection of research results by routinely and automatically recomputing the results of all past research projects. A research group can verify the completeness and functionality of a software release of disparate research projects by recomputing all the results. For researchers programming in Java, a script can automatically derive Internet enabled reproducible documents from the standard products of a researcher: the written paper (e.g. in LaTeX), the programs (Java for Internet use), and a command script (e.g. a makefile or a shell script).

INTRODUCTION

The idea of reproducible research

Is the Internet nothing but a huge library? A stack of dusty documents and illicit images? Maybe today, but for computational researchers it can be much more: it can be a globally shared laboratory. I advocate that computational researchers – chemists, physicists, engineers, applied mathematicians, astronomers, geophysicists, or even meteorologists – change

[1]**email:** Author has no known email address

their practice and philosophy of publishing. Reading new material may never again be done in an armchair in front of the fire place. Instead research publications are to be explored by the researcher as well as by a researcher's computer. The researcher reads the article, a *mere* advertisement and exposition of ideas. The researcher's computer recomputes the articles results and verifies that the implementation, the details, the software, is complete and readily available to the reader.

I do not suggest that all researchers submit to some complicated standard of programming or that all researchers have to add cute interactivity to their publications. My goal is simpler, more general, and more easily achieved. I suggest that researcher make their computational results reproducible.

Reproducibility is the crux of scientific work and the key to the efficiency and productivity of the modern age. It is what Newton called *standing on other's shoulder to see further*. Scientific results own their power to the principle that they apply anywhere and anytime and consequently that anyone can reproduce and apply them. A mathematical theorem or a simple computational algorithm can be described exhaustively in the summary of a printed journal.

Computational research is a form of experimental science. A given data set is a problem, a given algorithm expresses a scientific model, the application of the algorithm to the data is an experiment. In publications, researchers present the idea of the algorithm and document the usefulness of the underlying model with a few results. While our computational experiment is fully reproducible in theory, in practice our published results hardly ever are.

The reproducibility of computational research experiments, however, is flawed. Modern computational experiments usually depend on a multitude of implementation details and parameters that cannot be communicated efficiently in a printed publication. A researcher Newton that reads about some computational results by a researcher Galileo might spend about as much time to reimplement and verify Galileo's computations as it took Galileo in the first place. Consequently, in computational research the traditional publication is simply advertisement for the research stored in implemented algorithms.

But the computer that allows us to do such complex computations in the first place also allows us to preserve and communicate them efficiently. Pseudo-standards develop in computational communities for programming languages (Matlab, C, or Fortran) and data formats (SEGY, CDF). Large disks, CD-ROMs and the Internet facilitate easy data exchange and storage. The current bottleneck is to preserve and communicate the computational functionality of a project's pieces: The commands that combine the collection of programs and data files to an organic, functional entity and produces the desired results.

I suggest that research is published as electronic reproducible documents that combine a project's software and its scientific document and offer for every result three standard commands to any reader:

- **View** displays the result and offers it for scrutiny.
- **Build** recomputes the result from a given set of source files (such as data files and computer programs).

- **Info** lists any information important for the reproduction of the result, e.g. a time estimate for the recomputation of the figure or a commercial software needed.

If the recomputation of the result file creates intermediate files an additional **Clean** button offers the reader to delete all intermediate files. In a simpler version, the **Build** command deletes all intermediate files after it recomputed the result files.

In an early paper (Schwab and Claerbout, 1996) I describe the solution of my laboratory (Jon Claerbout's Stanford Exploration Project) based on C, Fortran programs, LaTeXand makefiles. A variation of this system is used by William Symes at The Rice Inversion Project (1998). David Donoho (1996) developed a similar concept for his Wavelet research using Matlab. Other researcher (Nagler, 1995) (need NASA citation) recognize the need for reproducible research as well.

On first impression, reproducibility must sound ridiculously altruistic and inefficient to a researcher eager to churn out publications. In my experience, the author of a reproducible document reaps the largest benefits from it. Every researcher has sad stories how the own programs and projects become alien and unaccessible over time. The proposed reproducibility commands facilitate a frequent and automatic rebuilding of any and all past research projects. with a minimum effort the research can maintain a portfolio of former projects well-oiled and ready for continued use. The researchers size of software library and his confidence in the library's quality soar. In that sense, an author is the first reader of her own research.

Furthermore, I claim that a researcher can easily add the three reproducibility commands to the standard computational experiment. During active development, the researcher ceaselessly repeats the computation. Given a set of short shell scripts or makefile rules, (see below for examples) the author needs to adhere to a few naming conventions to implement the three reproducibility commands. The implementations of the reproducibility commands may vary, but I suggest that the functionality of the three basic commands – the reader interface – is universal and standard.

Is the computational research community going to revolutionize once we distribute reproducible research? Maybe. I believe, every researcher would like to be more productive by using another researcher's software. In my experience, reproducible research documents using Fortran, C, and LaTeX, and GNU make improved collaboration among the researchers of my laboratory. However, the distribution of our documents and software to sponsoring industrial research laboratories on CD-ROMs failed since it required a rather difficult installation step on a UNIX system[2]. A reader who considers using a third-party software desires a stepwise adoption that lets him continuously evaluate the cost-benefit the software offers. A time-consuming download and installation step of a software package of dubious quality and documentation, is unattractive. A third-party Java software package is much more attractive: Java's integration in Internet browsers enables the reader to test the software actively before downloading it. Java's portability eliminates the installation step. The test establishes that the advertised research is available, complete, and correct.

[2]GNU and Linux (especially Redhat) have done much to improve the installation of UNIX software.

An author needs to offer software in self-contained units that integrate into a reader's environment. The traditional software categories by distribution were

- a project's private, highly individual files (Usually the core of the research project).
- an author's standard repertoire of software tools that he shares among research projects and possibly among laboratory colleagues.
- an author's tools that are common among readers of the author's research community.

To distribute their software, authors need to judge their readership and package the software correspondingly. Using Java, a reader's browser downloads the programs it needs to do the reader's bidding. The author does not need to prepackage the various programs.

In summary, I believe that the combination of Java and reproducible research documents may change the way the computational research community publishes. Java invites reader to explore programs without committing them. Reproducibility commands standardize the exploration and organize a research project in functional units. Both technologies – Java and reproducible documents – can be used independently, but in their combination they offer opportunities to any researcher seeking an audience.

EXAMPLE

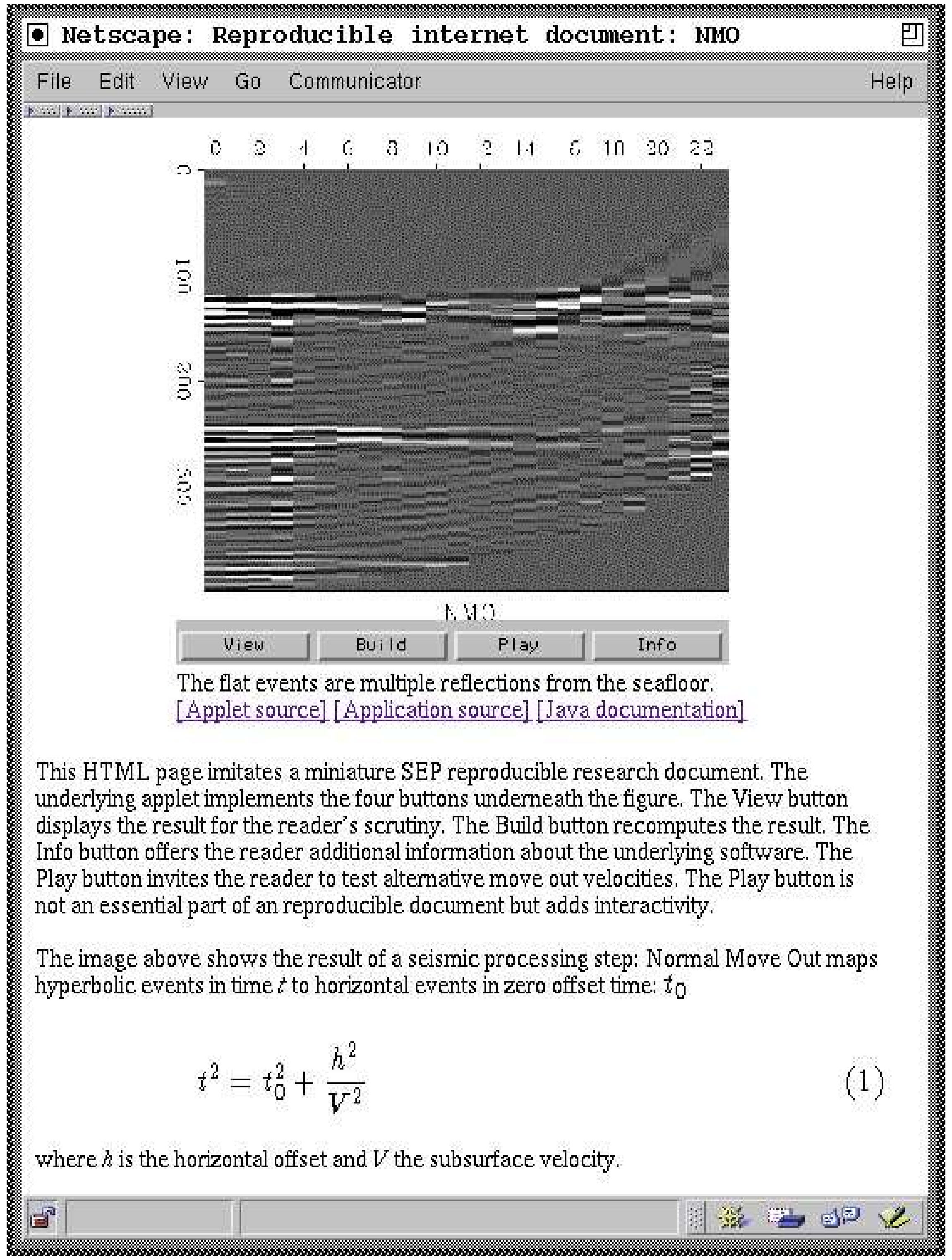
The HTML page http://sepwww.stanford.edu/sep/matt/join/ijDoc/reFig/reFig.html includes an applet that displays and reproduces a standard seismic processing result: a normal-move-out of CMP gather dependent on a given constant velocity. Figure **??** displays a Netscape browser accessing that page. To successfully access the page, your browser must run Java JDK1.1 (I tested this successfully with Netscape 4.0.5 and HotJava 1.0. I am told that Internet Explorer 4.0 also supports Java 1.1.). In an Java enabled browser, the buttons in the Figure caption execute the reproducibility commands.

The HTML page and the underlying Java applications are a prototype. Below I discuss a few implementation details of this prototype. Finally, I argue that such Java-enabled reproducible pages can be derived automatically if the underlying program is coded in Java.

Dear Reader: the rest of the this article discusses details of a reproducible document implementation. If you were only interested in the concept of reproducible documents, you may want to skip the rest.

COMPONENTS

To implement the prototype of a reproducible Internet document I had to supply the HTML page, the Java applet that implements the reproducibility commands, the application program, and the Graphics software to display the results.



The flat events are multiple reflections from the seafloor.
[Applet source] [Application source] [Java documentation]

This HTML page imitates a miniature SEP reproducible research document. The underlying applet implements the four buttons underneath the figure. The View button displays the result for the reader's scrutiny. The Build button recomputes the result. The Info button offers the reader additional information about the underlying software. The Play button invites the reader to test alternative move out velocities. The Play button is not an essential part of an reproducible document but adds interactivity.

The image above shows the result of a seismic processing step: Normal Move Out maps hyperbolic events in time t to horizontal events in zero offset time: t_0

$$t^2 = t_0^2 + \frac{h^2}{V^2} \tag{1}$$

where h is the horizontal offset and V the subsurface velocity.

Figure 1: The shown HTML page uses a Java applet to include a reproducible figure and its push-buttons. $\boxed{\text{nmoHTML}}$ [NR]

HTML PAGE

The HTML page describes the research topic, is specific to the research project, and can usually be automatically derived from the author's preferred text file format (e.g. a LaTeXfile).

The Java applet tag embeds the reproducible result in the HTML file. When the HTML page is viewed by a Java enabled browser, the applet tag instructs the browser to download and execute the Java program `ReFig.class`. The browser finds the Java program relative to the `codebase` directory at the computer serving the document. The Java program is an *applet* since extends the Java applet class. Only applets can be executed by a browser.

JAVA APPLET

The Java applet displays the Gif version of the result figure `nmo` and three buttons in the caption of the figure. I use the standard Gif format for the figure to guarantee a robust display of the document during browsing. The buttons offer the standard reproducibility commands to the reader.

The applet can download and execute a small set of underlying Java applications: The applet can download a given SEP data file and store it as a multi-dimensional Java array. The applet can invoke an application that processes a SEP data file that was downloaded earlier. The applet can convert a downloaded SEP data file into the pixel representation of the graphics package and instruct that graphics application to display the data. The downloading, conversion, and display instruction are identical for any result file. The invocation of the processing application will of course vary from project to project.

View

The View button starts a Java graphics application (see below about ImageJ), that displays the result figure in an independent window. The graphics application let's the reader explore the figure (zoom, scale, smooth, sharpen and any other standard ImageJ process your version of ImageJ supplies). To display the figure, the Java applet (run by the browser) contacts the document server, downloads a SEP data file that contains the result, converts the data to its internal pixel representation for floats, and displays the pixels.

The displayed image is not the Gif image displayed in the web browser. But both images were derived from the same result data file. I will have to say more about this in the section about the Java application.

Rebuild

The Rebuild button contacts the document server, downloads the SEP data file that contains the input, downloads the Java application to processes the data, reprocesses the downloaded data, converts the recomputed result file to graphics pixels, and displays the data. Ideally the processing application shows the reader a progress bar that indicates how much longer the reader will have to wait until the image is displayed. Ideally, the application would indicate what computations it currently executes.

Info

The Info button offers the interested reader more access to the software he just experimented with. The pop-up info window may include a link to the applet source code, a link to the Javadoc documentation of the author's software, a link to a web page that offers the software for downloading, a link to the author's home page. Furthermore, the info window may state a time estimate for the reproduction of the result image. I tend to place some of the links directly in the Figure caption.

In contrast to a reproducibility scheme with unrestricted access to the file system (see SEP's makefile implementation), the applet version of the reproducible document does not need a clean or burn button since there are no local files. This might change when reproducible documents start taking advantage of possible access to the local file system. An interactive result (such as Figure **??** may include additional buttons, or it may invoke additional graphical user interfaces.

JAVA APPLICATION

Reproducible documents are a reader interface for research and their implementation does not require a specific choice of computational tools and conventions. A simple black-box description of input and output files is all that is needed: Any result is usually represented by a single output file (usually a data file). The result file might have various display versions such as a postscript file or Gif file that are all trivially derived from the output file. A command (which could be a composite of several program invocations) computes the output file from a set of input files. The input files are usually data files and programs. I refer to the set of programs that compute the result files from those source files as the application. Note that the application may very well include the compilation of programs.

In the past we considered the data result file an intermediate file. Today, I feel that this data result file is the actual result and that the various graphics representation are trivially derived from it. By making the data result file central, the reproducibility becomes far more flexible with regard to various output formats such as postscript, dvi, or HTML documents. The shift in philosophy is important and has various practical consequences.

GRAPHICS

An ability of an image to communicate the important features of a large data set makes graphical display of results an essential part of many computational research fields. Since I want to distribute my graphical results on the Internet, I chose a Java graphics package.

I am not an expert on graphics software, but let me list my ad hoc requirements for a Java graphics system. The software needs to display multidimensional arrays as grey-scale or color rasters. Simple graphics instructions draw lines and other elementary geometric figures. The package would include simple interactivity such as the zooming of image details and the display of image sequences as movies. Furthermore, the graphics software should allow custom plug-in processes for the display needs of particular images. A sophisticated volume renderer would offer interactive exploration of large, high-dimensional image cubes (similar to Rickmovie). Besides the data display, the graphics package would allow a flexible annotation of the images with titles, axes, and labels. A set of simple classes allows the authors to build simple interactive graphical user interfaces to their applications. These various goals could be accomplished by a combination of tools, but a consistent style of methods and parameters would make a single package desirable.

Since I lacked a suitable graphics package, I decided to use *ImageJ* (Rasband, 1998) of the NIH (National Institute of Health), rather than implement my own. NIH distributes a very successful non-Java graphics software and I trust the Java version will be further maintained and improved in the future. I decided for ImageJ since it offered a set of image processing routines and anticipated user extensions. Using a third-party software, of course, required that I compromise a few of my lofty goals.

ImageJ

ImageJ requires a browser that implements JDK 1.1. ImageJ accepts various data input formats including arrays of floats. Furthermore, ImageJ offers a set of simple interactive image manipulations such as zoom, edge detection, image brightness, etc. Users of ImageJ are invited supply additional plug-ins that extend ImageJ's capabilities to a specific data format or special processing tools. The pages

```
http://sepwww.stanford.edu/sep/matt/join/ijDoc/ipDem/ipDemo.html
http://sepwww.stanford.edu/sep/matt/join/ijDoc/imageJ/imageJ.html
```

show two examples of ImageJ.

Currently, ImageJ does not offer all the graphics capabilities that I want, but Wayne Rasband, the author of ImageJ, promised to add more features soon. Currently, ImageJ does not support movies and image annotation. ImageJ displays an array of pixel values. It does not have a notion of axes beyond the number of pixels along each axis. Wayne Rasband suggests to add axis information as a plug-in. ImageJ's distinction between an image's displayed data values and its auxiliary features such as axes or buttons, must simplify ImageJ's data manipulations. Overall I am glad that I did not need to write a graphics package from scratch. For more information on my experience with ImageJ see the Appendix.

AUTOMATIC DOCUMENT GENERATION

I have not developed the tools to create reproducible Internet documents from my standard research files: Java programs, LaTeXdocuments and makefiles. However, I believe, the automatic creation of reproducible Internet documents is possible without incurring hardly any additional work for the author who programs in Java. In general, programs in other languages are not executed by browsers since the browser is unable to protect the reader's computer from computer viruses and other possible attacks.

When I implement a computational research project I usually create a LaTeXdocument, a Java application, a few data files, and a makefile. The makefile integrates the various pieces into a functional unit. For example, the makefile stores the commands to compute result files from input files or converts result files to various image formats for inspection and inclusion in the LaTeXdocument.

HTML

To generate an Internet version of the research document, The script latex2html (Drakos, 1996) translates the original LaTeXversion to an HTML document. The script replaces every LaTeXmacro that includes a figure (an activeplot macro at SEP) by an applet tag. The figure name is an argument of the LaTeXinclusion macro; latex2html may list the figure name as a parameter of the applet tag.

Application

The Java applications that process the input data to the result are invoked by the applet. The author needs to ensure that none The applications are restricted by the browser's Java security manager when executed within a browser. Consequently, applications cannot use Standard in, out, and error streams. The application can only access files from the original document server and do not have access to the clients local file system. Acme (ACME, 1998), a Java software company, claims to have a general approach to ensure that all applications can execute in an applet. Recent browsers include security managers that can be adjusted so that many applet restrictions do not apply to trusted applets. Finally, the Java class files have to be available in the package hierarchy the applet tag's codebase points at.

Applet

A script may create the Java applet that displays the result figure and implements the three basic reproducibility commands for a result figure. As stated above, the applet downloads, processes, and displays data files.

The code for downloading and displaying a data file is independent of the particular result file. The result file name and the figure size can be extracted from the plot macro of the LaTeXdocument.

The applet's Info method The script can probably extract the author's name and email from the LaTeXdocument. The author may have to supply the software home page. The script would be able to state the figures reproducibility from its classification as non-reproducible, conditionally-reproducible, and easily reproducible that SEP makefiles include.

The application specific command invocations that generate the result file from an unknown list of input files pose the central difficulty in the generation of reproducible Internet documents. These commands depend on the specific application and we can assume little about their specific form. Traditionally, I stated these commands in a makefile rule for each result. There are several potential mechanism to make these commands available to an applet.

Makefile extraction

A script could extract the commands necessary to recompute a result from the author's makefile. Since Java main methods can be invoked with a string argument just like any other method. a script can translate the makefile command into an applet's method invocation. I expect that we can write a script that automatically translates even fairly complicated makefile rules to applet methods. In my experience, object-oriented Java applications tend to be accompanied by simple makefile rules. Things that were done by stringing together procedual programs in pipes are implemented as object methods and are invoked within a program.

Jake: a Java implementation of make

A Java version of make would elegantly integrate makefiles and Java. If we had a Java version of GNU make – I would have to be called jake –, the applet could download the makefile (a simple text file) and interpret the rules accordingly. It could check dependencies and find the classes and inputs that it needs. Makefile rules would use relative filenames so that the browser could locate the resources relative to an applet's document base or code base. Since Java has its own mechanism to up-date Java classes from Java source files, we only need to maintain the rules that compute result files from source files.

We may prefer a Java make implementation for reproducible research that deviates from the standard GNU make. Program installation, the traditional task of GNU make, and document maintenance, the goal of our jake implementation, share the dependency and up-to-date mechanisms. Particularly, built-in commands for jake would list source, intermediate, and result files of a given target.

Since jake is written in Java, it is easily downloaded and installed. A browser may use jake without a reader of a document ever knowing about it. On the other hand, inexperienced authors of reproducible documents may prefer dealing with a well-documented software standard such as GNU make or a true Java version of it.

No more make

Makefiles are based on rules or logical programming, which can be difficult to write and debug. Before the dawn of the makefile, programmers used simple scripts that stored a sequence of complex commands. These shell scripts execute *all* commands when invoked independent of the files up-to-dateness.

How about an object-oriented replacement of the makefile concept? A Java interface `Target` contains the state of up-to-dateness, the list of dependencies (targets the list depends on), and the method to up-date itself (build rule). Possibly, a method to free all intermediate resources (such as files) the target creates when rebuilding itself (clean rule). As mentioned before, Java source and class files field such an up-to-date mechanism that allows the Java interpreter when invoked with the check-source flag to ensure up-to-dateness of the Java class file.

By the way, I believe that for Internet applications the traditional make concept of a target as a file should be generalized to include anything that can implement the described Target interface. For example, URLs, sockets, or processes might implement the interface.

In the case of reproducible documents, the result file would need to implement an interface that extends the target interface. A build method would simply invoke the up-date method of the target. A view method would represent the result file graphically. Which graphics method is used would depend on the target's environment (Do I have a customized view method? Can I invoke a default display?). A clean method would invoke the clean method of the target and its dependencies. Interestingly, an object-oriented reproducible document implementation would not include a burn rule: the result target is the natural object to up-date itself. Once removed the rules to recreate it would be removed as well. Finally, an reproducible result object could include an info method that informs the reader about such mundane things as the result's author or the ease of reproducibility.

Given such target objects we could create standard makefiles and applets to offer access to reproducible results through the traditional command line or through the Internet. To create a makefile, the script would simply query the target about its dependencies and formulate them as a rule. To create an applet, the script would implement buttons for view, build, and clean that invoke the result target's corresponding methods.

Reproducible documents on the Internet

So what is different about offering reproducible documents on the Internet? Obviously, we can reach many readers effortlessly. The readers can invoke our documents reproducibility commands and can download our source files. Theoretically, an author can give a reader as much functional access to a document as she has herself. Almost ...

A reproducibility command on a local computer can create a file on the local computer system; In the past, a reproducibility command executed by a browser could not. For security reasons, browsers restricted an applet's access to resources on the client machine. Particularly, an applet was unable to write or read a file to the local file system. Consequently, applet's were restricted to in-core computations and a reproducible research document did not need a clean rule. Recent browser versions (e.g. Netscape 4.0, HotJava) allow a reader to grant applets access to local files. The reader indicates trusted web sites in the browsers security settings and the browser verifies an applets origin when loading it from the web. My example of a reproducible result for the web does not exploit this new flexibility in browser security.

A reproducibility command on a local computer ensures the up-to-dateness of all dependencies by recomputing what is out-of-date. An applet is traditionally limited to downloading and processing existing files from the server. Usually an applet does not verify that a given file on the server is up-to-date or invokes a different command when the file is out-of-date. I believe that such a mechanism is feasible, especially if we had a Java mechanism to up-date targets. The current web prototype does not offer an up-to-dateness check. Instead the user has the simple choice between inspecting the finished result or recomputing it from scratch.

A process issued on a local computer enjoys a larger bandwidth when accessing data on the server than a process issued by an applet executed by a remote browser. We can ease the effect on large computations by distributing separating the downloading and the interactive execution. Either as CD-ROMs or by ftp from a data server. The data will then be local on the client machine during the execution of an applet (This again requires that the applet has access to the local client file-system). The distribution of large data sets still inhibits my vision of an effortless access to remote reproducible results.

A few more musings ...

- How does reproducibility differ from interactivity? Interactivity of a figure does not replace basic reproducibility. Interactivity helps a reader to understand the underlying algorithm by exploring an example. Reproducibility recomputes a given result without any reader input. Reproducibility ensures the basic functionality of a given software and delivers a working example. Not more, not less.

- I resized the Gif produced by our make rules by hand (using xv and the image-size button) because I thought the Gif images were too big (too many pixels). I wonder how this can be automatized in our standard processing. The pixel size I chose was 333x270.

- My content handler for SEP data cubes is not yet implemented for general SEP data files.

- Before distributing SEP data files on the web, I want to strip the header information to the bare minimum, e.g. (`n1, n2, d1, d2, o1, o2, esize` and possibly `in` and `dim`).

REFERENCES

ACME, 1998, Java software: http://www.acme.com/java/software.

Buckheit, J., and Donoho, D., 1996, Wavelab and reproducible research: http://playfair.Stanford.EDU/~wavelab.

Drakos, N., 1996, All about LaTeX2HTML: http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html.

Nagler, J., 1995, Coding style and good computing practices: http://wizard.ucr.edu/~nagler/coding_style.html.

Rasband, W., 1998, ImageJ: http://rsb.info.nih.gov/ij.

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

Symes, W., 1998, The Rice Inversion Project: Homepage: http://www.trip.caam.rice.edu.

APPENDIX A: IMAGEJ

Directory hierarchy

On my local machine, I want my home directory and my web directory to be identical. Any file in my home directory is also available on the web (possibly possibly by a password). I prefer to group my files by object, such as a research project on deconvolution, or my private stuff.

Since an applet only offers two entries to the server file system – the document base and the code base URL – all files relevant to that applet have to be named relative to these two entries. This is comparable to having a executable path that only accepts a single directory. Since my various projects in some cases overlap, I have to organize all files in a single, large file hierarchy. If the code base and document base arguments were to accept a directory path, I could organize my files in more manageable smaller object-oriented groups.

Given these limitations and preferences, here is the directory structure I came up with after some thought and experimentation:

```
ij:
gui:      io:       plugin:     process:
ijDoc:
plugs:    paper:    imageJ:     ipDemo:    reFig:    Fig:
```

The ij directory contains all the files I received from NIH. The ijDoc directory contains all the files I wrote. (I did not write imageJ and ipDemo but I modified them to a degree that I wanted to ensure that they are not being lost when I up-date the ij image software.) This allows for smooth up-dates of the NIH software and adheres to my principle to group files by object (this case authorship) not file type.

The subdirectory `ijDoc/plugs` includes all the SEP specific Java applications such as a simple handler of SEP data structures and the NMO application. The subdirectory `ijDoc/reFig` contains the html page that describes the NMO process and the applet that invokes the NMO example. The code base is defined as `../..` which is the relative path to the correct parent directory. All Java programs are named relative to that code base: so the full NMO applet name is `ijDoc.reFig.ReFig.java`. Keeping track of these names and the relative path name is the price I pay for an object oriented filing system compromised by a single code base.

The `ij.properties` file lists plug-ins and other resources that ImageJ should offer to the user. The `ij.properties` file has to be in the document base directory of the ImageJ invoking applet. I wish the name of that property file could be a parameter argument in the applet tag. A single HTML file could include numerous applets each with its individual property file. This would allow an author to guide the reader by customizing imageJ for each individual image.

ImageJ contains four methods to locate images and data. All of them are found in the File pull down menu. The `New` button creates a synthetic data set (e.g. a white image) and works as soon as ImageJ works. The `Open ...` button gives you a file dialog window that allows you to locate an image file (gif, jpeg) somewhere on your local file system. I am not sure how the button works, if you execute the applet remotely. For remote execution you have the `Open` button without dots. When pressed the button offers a second menu with figure names: the author of the applet defines the figure names and the path to these figures (again gif or jpeg) in the ImageJ's ij.property file. When run locally the images need to be in an subdirectory named `images` relative to the applet's document base. (I wish that would be a bit more flexible). Finally the `Acquire` button offers the user a variety of accesses and image formats: The `URL` option prompts the user for a URL, the `Raw` option gives the user a file dialog box to locate a file with floats. It will further prompt the user for pixel width and height.

ImageJ modifications

I felt that I had to modify a ImageJ slightly. I believe that all of these changes will be part of future releases.

I added a scale plug-i for the FloatingProcessor class. I added an ImageJ constructor that accepts an applet reference. I store the applet reference in the Info class. Consequently, imageJ can acquire applet parameters and values. (I hear Wayne Rasband fixed that in the newest release of ImageJ).

I suggest future ImageJ versions should make the `ij.properties` file an applet parameter, so that any applet can have its individual list of plug-ins and the author of an applet can name the properties file.

Software I wrote to complement ImageJ

Here are the Java applications I wrote to extend ImageJ to SEP data and applications. Some of this is very preliminary but could easily be improved. The source code is and API documentation is at:

```
http:/sepwww.stanford.edu/sep/matt/join/ijDoc/plugs
http:/sepwww.stanford.edu/sep/matt/jest/jdocs/api/Package-ijDoc.plugs.html
```

- `SepCubeHandler` implements a Java Contenthandler for SEP data cubes. Given a URL connection to a SEP data cube the SepCubeHandler reads the dimensions of the header information and loads the corresponding floats into a Java array.
- `SepCubeHandlerFactory` implements a Java ContentHandlerFactory for SEP data cubes. It creates a SepCubeHandler when it encounters a URL with file suffix `.H` or Mime type `application/x-sepcube`. It is currently not used since the security manager does not allow applets to execute ContentHandlerFactories.
- `SepCube2FloatArray` converts a two-dimensional Java array to a one-dimensional array. The conversion is necessary since my SEP applications expect multi-dimensional Java arrays while ImageJ expects a one-dimensional array.
- `Nmo` applies hyperbolic move out to a two dimensional Java array.
- `Sep` downloads an SEP data file (using SepCubeHandler), applies hyperbolic move out to the loaded data (using NMO), and displays the result using ImageJ. This class implements a Plug-in.

Miscellaneous

The nondescript file ImageJ.pif is a shortcut file for the Windows desktop. I removed it since I run Linux.

ImageJ thinks of the data it loads as files (FileOpener, FileInfo) where my plug-ins treat them as Input and Output Streams.

I hope ImageJ will soon take advantage of Java 2D and 3D. ImageJ uses a depreciated Java event model. Here is the warning from my Java compiler:

```
826 offenbach> /usr/local/java/bin/javac -deprecation ijDoc/reFig/ReFig.java
ijDoc/reFig/ReFig.java:48: Note: The method boolean action(java.awt.Event, java.lang.Object) in class java.awt.Component has been deprecated, and class ijDoc.reFig.ReFig (which is not deprecated) overrides it.
  public boolean action(Event e, Object arg) {
                  ^
Note: ijDoc/reFig/ReFig.java uses a deprecated API.  Please consult the documentation for a better alternative.
```

At the NIH website (http://rsb.info.nih.gov/ij/), I could not download the imageJ version 58 (which is compatible with Java JDK 1.0.2) since the files are not zipped into a single file. I did the prudent thing and up-graded my browser navigator to version 4.0.5 that supports Java JDK 1.1.

The web site offered two packages: for Mac users or Windows users. It was unclear what UNIX users such as I should do. I downloaded the window version of ImageJ unzipped it, and recompiled all classes by compiling `javac ij/ImageJ.java; java ij.ImageJ;`. It worked like a charm.

To test it I ran ImageJ stand-alone. I needed to download the Lena test image. I suggest to include that test image in the appropriate example that come with ImageJ.