LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T950057 - 00-E** 9/29/95

AVS Software Standards Guide

James Kent Blackburn

Original distribution of this document:

Albert Lazzirini, Hiro Yamamoto, Dennis Coyne

Robert Spero, Lisa Sievers

Rolf Bork, David Barker

Andy Kuhnert

CaRT

California Institute of Technology
LIGO Project - MS 102-33
Pasadena CA 91125
Phone (818) 395-2966
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu
WWW: http://www.ligo.caltech.edu

LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

- LIGO -

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T950057 - 00-E** 9/29/95

AVS Software Standards Guide

James Kent Blackburn

Distribution of this draft:

Albert Lazzirini, Hiro Yamamoto, Dennis Coyne

Robert Spero, Lisa Sievers

Rolf Bork, David Barker

Andy Kuhnert

CaRT

California Institute of Technology
LIGO Project - MS 102-33
Pasadena CA 91125
Phone (818) 395-2966
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu
WWW: http://www.ligo.caltech.edu

TABLE OF CONTENTS

1 INTRODUCTION

This technical document is intended to layout guidelines in choices of software languages, software style, software documentation and source code configuration control. The individual LIGO software developer, through the use of common protocols discussed in this document, will produce software that is easy to read and maintain, that is well understood and extensible and that has long lasting value to the project. This document is not meant to be a detailed tutorial or the definitive source of knowledge on the subject. The introductory chapter of the second edition of Numerical Recipes provides the interested reader with other aspects of this subject.

This document will also try to present some of the concepts and current software development tools available under Unix that may assist the software developer in staying close to the less than sharply defined boundaries of software standards. These include the importance that high level languages such as FORTRAN and C have on style, the significance of compiler flags, and the use of poorly documented library functions such as random number generators[1].

Since the aim of this document is to describe software standards for use in AVS, it is only natural that some discussion of software development under the AVS environment be included. Among the points to be covered are a brief overview of what the AVS environment brings to a large software program and what are the interfaces provided by AVS to the traditional high level languages such as FORTRAN and C and how these may be carried forward to include other environments like Matlab and Mathematica. Examples of AVS modules in C and FORTRAN for random number generators are also given in the appendix.

2 SCOPE OF A SOFTWARE STYLE GUIDE

Computer programs are based on source code developed using computer languages such as C and FORTRAN. The reason for using high level computer languages is to make the source code readable by humans, not by machines. This is especially true in a research program such as LIGO where there is little doubt that either through new found knowledge or even as the result of bugs introduced by the complexity of the models, any source code written will at some point need to be revisited either by the author or someone who has taken on the original author's responsibility. To make the task of reading source code as easy and efficient as possible, it is advantageous to introduce and even to demand software coding practices from the very beginning of a project.

But readability is not the only reason for having software style guides. Another even more compelling reason revolves around the issue of standardization. The compilers being used today are not the compilers to be used in the future. Nor are the software and hardware vendors that are currently in place guaranteed to be the primary platforms used even six months from now. By following the guidelines of standards as closely as possible, the tendency to develop backwards compatibility into computer languages will provide forward compatibility for any software developed today.

Still, the use of standards is not always enough. Issues of style can increase portability across current platforms that may be a part of a probable user community of any significant software tool.

---

1. A brief discussion of random number generators is given in Appendix 3.

An example may serve to make the point. The Sun FORTRAN compiler initializes all local variables to zero upon entry into any subroutine or function. In order to do so the Sun FORTRAN compiler must make a table of all variables that appear in the subroutine or function and initialize each variable to the appropriate bit pattern for zero based on the data type of the variable. This happens before actually executing user-specified computations. The advantage of this is that the careless programmer can neglect to set a variable that may be used as a simple counter or as part of a more complex logical algorithm and the program will still work correctly. Even the ANSI standard for FORTRAN does not address this issue. Now here is where the issue of software style becomes important. Most high performance FORTRAN compilers for ultra-fast computers do not take the time to initialize variables the user. These computers were designed to work with extremely large data sets where the time required to initialize these large arrays becomes a major performance issue. These compilers rely on the software developer's understanding, comments and style to make debugging such pitfalls manageable. The software is more easily ported to these platforms if it has comments documenting the use of variables and their initial values.

3 AVS OVERVIEW

Advanced Visual Systems or AVS is a modular software system driven by a 'point and click' mouse interface. The package provides a marvelous set of tools for manipulating multi-dimensional sets of data using very well thought out data structures. AVS modules are functionally equivalent to subroutines in FORTRAN or void functions in C. Modules have a unified communications mechanism among themselves known as AVS ports. Each module will have, depending on its design, a set of input and output ports used to pass data along from one module to the next in a hierarchical scheme known as an AVS flow network. In addition, modules may possess any number of parameters which are controlled by the user through easy to understand graphical widgets such as dials, sliders and text fields. The modules are stored in a graphical palette, much like a library and can be dragged down to the flow networking editor where intercommunications are established using the mouse alone.

AVS supports passing many of the standard data types, such as byte, integer, real and string, between modules. In addition, AVS has field, colormap, geometry, pixelmap, UCD and User-Defined data types for exchanging data. In most cases the most useful of these will be the field and the User-Defined data types. Of these two data types, the field has the advantage of already having modules supplied by AVS for reading and writing the data to file. The field data type is a general representation for an array of data. It also has a self contained description of the data. The field uses either an implicit or explicit mapping of the data elements to coordinates. This provides a very general and very powerful representation for data. The AVS field data type is analogous to a C structure and is defined in the <avs/field.h> header file. Because of this one-to-one relationship between the AVS field and a C structure, the field data type is most easily used from the C language by a software developer fluent in C. However, AVS does provide a set of subroutines to access and manipulate any desired elements of the field data type from FORTRAN. Using the FORTRAN calls to access the field adds an extra burden on programmers revisiting the source code since they would now need to understand the purpose of the calls to the AVS FORTRAN library. As is always the case when using non standard library routines, the software developer should fully document through comments the purpose of all calls to the library routine.

The AVS module represents one procedural unit of computation. Each module takes up a process ID slot in the Unix kernel[1]. Too many modules can fill up the available process slots in the kernel and cause the flow execution of the AVS network to fail. To avoid this, modules should not be so minimal in their function as to require many modules to actually accomplish something useful. For example, it would not be very practical to have a module for subtracting two integers. On the other hand, if too much functionality is placed into a single module, such as having the entire End-To-End Model as a single AVS module, then it would have too specific utility to be used in other applications and any variations of the Model would require re-construction. So modules should be unique enough in there functionality to be useful to multiple applications.

Software developers can easily develop specialized AVS modules using the C or FORTRAN languages. This basically involves generating AVS "wrappers" around C functions or FORTRAN subroutines. The "wrapper" is responsible for setting the AVS module name, creating input and output ports for the module, creating parameters for the module, setting the computation function for the module, and performing any additional initialization that is required. AVS has its own error handling mechanism which includes an error logging facility. It should be used instead of print statements in the body of the source code. Debugging of modules is also supported using the **avs_dbx** shell script On a Sun workstation using Solaris, the actual command to be used for debugging is **avs_dbx -debug debugger *module*** from the directory where ***module*** is located. For more information on AVS and developing modules in the AVS environment see the AVS documentation[2].

Any character strings of text printed to standard out by an AVS module will appear in the shell that was used to start up AVS. To avoid having to much information in that shell's window, it is recommended that a verbose parameter be provided to the AVS module that is an integer represented by a slider widget that ranges from 0 for no output printed to some maximum positive integer for print everything. This could prove to be a useful debugging feature in modules as they are propagated from the developer to the user.

Dynamic memory management is an important aspect of any AVS module. This is because the AVS module exists as a Unix process. Each time the module is run it should free up any memory that was previously allocated and then allocate the required memory again. This will prevent memory leaks from occurring when the module is repeatedly run with new parameters. AVS provides routines for dynamic memory allocation and these should always be used instead of the Unix equivalents.

4 SOFTWARE STYLE

This section addresses issues of software style and is intended to provide some guidelines to the various topics that bring 'added value' to source code. Among the key topics are readability, portability, structured programming, the use of popular high level programming languages such as C, FORTRAN and C++ in the AVS environment, along with a discussion of AVS software develop-

1. It is possible to place multiple modules in the same process. See page 4-27 of AVS Developer's Guide.
2. Recommended in the following order: AVS Tutorial Guide, AVS User's Guide, AVS Application Guide, AVS Developer's Guide, AVS Module Reference

ment. The software developer should consider these topics when writing his or her own software so that others may also benefit from that software.

4.1. Software Readability

There are many common sense approaches to software readability. The most highly promoted of these is the use of comments. Using comments does not mean to 'spray' one's source code with non-compiled words. Comments should be viewed in a systematic way. There should be a header to the code where a set of descriptors like **program name, author, creation date, modification history** and **purpose** are listed[1]. These can be and often are standardized across a project. Comments should also be used to describe formulae which should include references. Any use of library routines should be documented clearly. Understanding why a non-standard library call is used is one of the most time consuming aspects of revisiting source code. The same is true of subroutines and functions. Comments should also be clear and concise. Any lengthy explanation should be made in some form of documentation[2] and referenced.

A second area where readability is important is in expressing complex, lengthy formulae. Instead of trying to reproduce an expression as closely as possible to its analytic form, it is best to program sub-expressions and combine them in steps using several compiler statements instead of using continuation statements in an attempt to represent the expression in its original form. This can have the added advantage of improving performance when sub-expressions appear repeatedly and can improve performance. Breaking up expressions if also important for FORTRAN source code where the ANSI standards restrict the statement length to 72 characters[3].

Avoid being tricky with programming statements. There is almost an attitude among C programmers that as much as possible should be packed into the fewest number of characters possible. Try to avoid this temptation; it may win first place in the most esoteric one line C program contest, but it won't make friends among colleagues trying to find the bug you left in the software. It is also very helpful to give variables names that are somewhat self evident. However, care must be used to avoid violating ANSI standards for variables. All C preprocessors implement a set of macros used to control compilation. These macros vary from one compiler to another but always begin with an underscore. In addition, C-FORTRAN code binding in Unix usually requires adding an underscore on globals such as function names.[4] So use underscores as the first character in C with care. ANSI FORTRAN 77 doesn't allow more than 6 characters in a variable name which makes code less readable. Most Unix FORTRAN compilers allow more than 6 characters in names so use this extension unless you are developing code that may have to run on many different platforms where the longer variable names may cause more difficulty than is gained through the readability of these longer names. Also most Unix FORTRAN compilers allow the use of lower case characters. This tends to improve the readability of code as well, though it is also a violation of the ANSI FORTRAN 77 standard. Case of characters can easily be changed if needed, so do not give different FORTRAN variables the same names while differing only by the case.

---

1. For a complete list of header keywords, see Appendix 4.
2. See Section 5 of this Documentation
3. AVS FORTRAN wrappers adhere to this standard implying that AVS runs on platforms requiring it.
4. See page G-50 of the AVS Developer's Guide for additional details.

4.2. Software Portability

The importance of software portability is to make software written by one person on one computer platform with its specific compilers usable by others on other systems. Portability at a future date is also an issue for software. The operating systems and compilers will change with time. AVS may not be the environment LIGO uses 10 years from now. By developing portable algorithms at the heart of AVS modules, it will be possible to migrate to the AVS successor with minimal effort. The primary tool for maintaining software portability is to follow the ANSI standards for computer languages. This will typically mean ANSI C and ANSI FORTRAN 77[1]. There are several compiler flags and tools available for determining ANSI compliance and portability. For Sun's FORTRAN, try the **-ansi** compiler flag. There is also a public domain utility which is highly recommended called **ftnchek**[2]. It checks much more than ANSI compliance and is very useful for finding programming errors. For C, the standard tool for checking portability of code is **lint**. It also is useful for finding errors. These tools have Unix man pages and should be reviewed by all software developers. These tools are only aids, the programmer ultimately is responsible during its development for the portability of the software. The software developer must fully understand the flow of the code, which will be discussed further in the next section.

4.3. Structured Programming

As a program executes, the ordering of the statements unfolds through the use of control statements like *if-else*, *do-while*, *do-continue*, etc. Structured programming is the use of control statements in such a way as to make the flow of a program visually apparent to the reader of the source code for the program. Well structured code is also easier to develop and to debug. It is not possible to design structure into programs that use goto statements and label statements. Code written in this style is routinely referred to as "spaghetti-code" and should be avoided unless absolutely required in certain very narrow and specific instances. Standard FORTRAN 77 has minimal support for structures similar to the while loops of other languages. However, with some creative thought on the placement of *continue* statements and *if* statements preceding a tame *goto*, it is possible to implement many of these structures.[3] Another important aspect of structured programming is to keep things modular. Design the software to flow into and out of blocks of code, not lines of code. Additionally, don't overload the functionality of subroutines and functions.

4.4. Programming Languages in AVS

The developer of AVS software modules has the freedom to choose between C, FORTRAN, C++ and other highly specialized software environments like Mathematica and Matlab. The C language provides the most natural constructs for working in AVS, but the others have advantages that may factor into the decision to use one over the other.

---

1. FORTRAN 90 compilers are not widely available.
2. See Kent Blackburn for the source distribution.
3. See Chapter 1 of Numerical Recipes (FORTRAN version) for examples.

4.4.1. Programming with C

AVS module development uses constructs such as dynamic memory allocation and data structures that are very familiar to the C programmer. Because of the similarities that exist between these two high level environments, the C programmer will experience the fastest learning curve when developing AVS modules with C. Additionally, many of the data types used in AVS can be directly manipulated from C since these AVS data types have a one-to-one relationship with C structures. However, C compilers have yet to reach the level of optimization that is currently available with FORTRAN unless the programmer writes her C code in the style of FORTRAN. In addition, the scientific analysis libraries available to C programmers are not as complete, nor as well tested as the scientific libraries for FORTRAN programmers. Still the C language is more structured and modular in its constructs. C code tends to be more portable as well, primarily because the C programmer is less likely to need to resort to compiler specific calls to support the needed algorithm.

4.4.2. Programming with FORTRAN

AVS includes a complete set of tools for developing modules in FORTRAN. This an extremely useful and important feature of AVS. FORTRAN compilers tend to be several times more efficient in numerically intensive software than other languages. There is also an extensive set of numerical libraries available to the FORTRAN programmer which have been thoroughly tested. Support for FORTRAN is also important to the End-To-End model since there is a significant amount of software that has already been developed which will be ported over to AVS modules. However, the handling of data and the memory management and the inter-module communications of AVS are addressed using AVS library routines. The naming convention for these libraries provide little more than a clue to the purpose of the call. So to make the intent of the code more clear, software developers should clearly comment the calls to AVS routines.

4.4.3. Programming with C++

Object oriented languages add a very powerful collection of concepts to high level languages.[1] Chief among these are the linking of functionality to the data, truly reusable code and polymorphism. AVS does not provide a clean interface to C++, instead module wrappers for the C language must be slightly modified to be compatible with a C++ compiler.[2] A crude AVS module, implemented as a C++ class is demonstrated in the AVS examples.[3] If you were *really* going to use C++ within a module, you would probably use the AVS functions as they are and develop the internal code of your modules using your own class libraries. This does not provide for a very robust implementation of C++ support and is therefore only recommended for modules where concepts of polymorphisms, inheritance, operator overloading and other C++ constructs will provide a major time savings in future source code development.

---

1. See Appendix 2
2. See section 8 of AVS 5 Update for complete details.
3. See the file $AVS_PATH/examples/cpp_example.cpp

4.4.4. Programming with Matlab and Mathematica

Both Matlab and Mathematica provide interprocess communications. For the AVS module developer, the direction of communications which will appear most frequently is to have the AVS module request that some analysis be carried out by the Matlab or Mathematica computational engine. But AVS does not support direct communications to the Matlab and Mathematica environments. To accomplish this, either C or FORTRAN must be used to cross the bridge. In the case of Matlab, the AVS module developer has the choice of interfacing with the Engine Library[1] through either C or FORTRAN. Using MathLink[2] which is distributed as part of Mathematica, the AVS module must communicate through MathLink C library functions. In all cases, deciding to use the Matlab or Mathematica computational engine should be based on a need to provide a solution quickly, because these tools require licenses and may not be available to others which means that your AVS modules may not be available to others. In addition, both Matlab and Mathematica may carry with them a performance hit that could be lessened by a compiled version of the code written in C or FORTRAN. Still having said all this, these are potentially useful interfaces to have available to implement from AVS modules.

4.5. Developing AVS modules

AVS modules are developed using the GUI driven **Module Generator** module found under **Data Output** in the **AVS Module Library** of the **AVS Network Editor**. Pulling down this module to the network flow palette produces a set of options for generating the C or FORTRAN wrappers for routines to be provided by the developer.[3] These wrappers are even complete with comment headers specifying the module name, author, and date created. There are four[4] separate sections within the wrapper. Each section is marked by a pair of comments that look as the following in C

```
/* ----> START OF USER-SUPPLIED CODE SECTION#...*/
/* <---- END OF USER-SUPPLIED CODE SECTION #...*/
```

and look as the following in FORTRAN.

```
C ----> START OF USER-SUPPLIED CODE SECTION#...
C <---- END OF USER-SUPPLIED CODE SECTION #...
```

It is crucial that developers place their lines of code between these comments. If not, when a modification to the AVS interface occurs, any lines of code outside the comments are lost! The types of interface changes that would cause this are changing the number of parameters, input ports or output ports. Also changes to the data types used by any of these. And of course, any time the user chooses the **Write Source** button for the **Module Generator**.

There are occasions when the developer of AVS modules will need to extend the functionality provided by the standard AVS interface. Two occasions for this are very likely. The ability to

1. See the MATLAB External Interface Guide for details.
2. See the Wolfram Research technical report, The MathLink Reference Guide for details.
3. See Section 2 of the AVS Applications Guide for a simple example or writing an AVS module.
4. FORTRAN wrappers actually have only three.

write code to address ports in the USER-SUPPLIED CODE SECTIONS is very poorly supported in AVS. To address these ports, it is necessary to modify the code written by the **Module Generator**, allowing for unique identifiers. If this is the case, comments placed around the lines of code that have been modified will act as a marker to others reading this code that this section must be reconstruct that by hand if the **Module Generator** is used to modify the AVS interface such as the parameters, ports or data types. In addition, make a backup copy of the changes in comment form in the closest standard USER-SUPPLIED CODE SECTION with detailed instructions on where the modifications should be recreated in the event that they are accidently lost through the use of the **Module Generator**. Any comments that surround these modifications must have a standard form to be useful in an automation script. For C code use the following

/* ====> START OF LIGO USER MODIFICATION TO AVS CODE SECTION */

/* <==== END OF LIGO USER MODIFICATION TO AVS CODE SECTION */

and in FORTRAN use these.

C ====> START OF LIGO USER MODIFICATION TO AVS CODE SECTION

C <==== END OF LIGO USER MODIFICATION TO AVS CODE SECTION

Be sure to implement these comments exactly. A future script can be developed to automate the process of making the needed modifications. Additional comments should also appear within these comments to expand the purpose of the modifications to the code originally written by the **Module Generator**. This is important since AVS may actually change the interface to the ports and parameters in a future version and the exact line of code used to make the modification would have no value, only their purpose could be used to make the needed changes.

5 DOCUMENTATION

The generic purpose of documentation is to furnish information and present proof of principle. This applies without exception to software development. Through comment statements in the source code, the proof of principle is documented for software developers. Through User Guides, on-line help such as the AVS module help standard and Unix man pages, information is furnished to the end users.

5.1. Source Code Comments

The lowest level of Documentation occurs in the source code itself in the form of comment statements. These comments serve to both document the software and to provide understanding to the flow of the program. As previously discussed, there should be a header section in the source code built out of comment statements that present the summary of the software. This should include, but not limited to, the name of the software, the author, the date created, the purpose and any modifications that have been made, including the name and date of the modification. All of these header comments will add to the understanding of the software when the source code is revisited at a later date or by others for the purpose of fixing bugs, adding new physics, or making general enhancements such as performance and user friendliness.

Comments should also be place within the body of the code. These comments are for explaining the flow of the code, describing the purpose of subroutine and function calls, and describing potentially confusing logic. The underlying physics or mathematics should also be referenced in the body, providing a complete conceptual view to the goal of the software.

5.2. AVS On-line Module Help

AVS has a standard interface for providing help on modules. This help is GUI based and is initiated by clicking on the small square button on the right side of each AVS module with the right mouse button. This brings up a panel that lists all parameters and ports for the module by their names and is color coded to their data types. At the bottom of this panel are a set of buttons, the first of which is labeled **Show Module Documentation**, and is used to get help in the style of a Unix man page. This panel also has several buttons used to control the behavior of the module that are not relevant to the present discussion. When the **Show Module Documentation** button is selected, a scrollable window appears which provides on-line help consisting of the module name, summary information, a general description, list of parameters, input and output ports, references to other AVS modules and any limiting features for the module. The developer of AVS modules must generate this on-line help for each module. This is done very simply by clicking on the **Write Man Page** button for the **Module Generator**. This will produce a file with the module's name followed by the extension ***.txt** which must then be edited to fill in the details. This is a simple text file so care should be taken to limit lines to 80 characters so as not to cause line wrap problems in the AVS help window. The value of this on-line help in furnishing information about the purpose of an AVS module is clear and its creation should not be neglected.To see the help page, the environment variable **AVS_HELP_PATH must** be set to the location of the ***.txt** help file.

5.3. User's Guides

Software documentation should also include User's Guides which provide both information about the use of the software and technical details justifying the formulation of the software. User's Guides should be more descriptive than on-line help pages about the use of the software by including details and examples. The User's Guide is the ideal place to present the models, formulae and numerical techniques that went into the software. It should also be the place to discuss any limitations and directions for future enhancements. This is also the type of documentation that is most likely to be found in paper form. As such it would naturally fall under the requirements of document control standards. Since it is useful to access such documents from a computer, they must be in a format that is available from Web browsers.

Using the Webmaker[1] program described under the LIGO Web pages, it is possible to convert Framemaker files into HTML documents that are viewable over the Web. It is very important to use the standard LIGO templates with Webmaker to guarantee proper interpretations of the Framemaker files. Another translator with is available takes LaTeX files and converts them to HTML. There are also HTML editors such as HoTMetaL[2] available for writing Web pages.

1. The URL is http://docuserv.ligo.caltech.edu/docuserv/home/convert_1.html (link from LIGO homepage).
2. To use HoTMetaL use the command sqhmpro from your LIGO Solaris Workstations.

Clearly, the route to take is to have User's Guides written as Framemaker files making printed versions available for document control purposes and then translate them to HTML format for use with Web browsers. Web browsers often have limited resolution and bandwidth when view large documents or documents with detailed diagrams so postscript versions of HTML documents should also be distributed by FTP from the HTML source and made available for downloading and printing.

5.4. Unix Man Pages

Template Unix man pages for modules can be generated from within AVS. This has the advantage of allowing Unix users to study the function of an AVS module without being in the AVS environment. AVS on-line help pages are pre-formatted into text files. This format is not suitable for Unix man pages. However, the AVS **Module Generator** can produce troff formatted files which can easily be processed through nroff to produce true Unix man pages. To do this the environment variable **AVS_MG_TROFF** must be set before starting AVS. This is accomplished by using setenv AVS_MG_TROFF from the shell which will be used to start AVS. Then when the **Write Man Page** button for the **Module Generator** is selected, the help file will be generated in the troff format. At the top of this file in the comments are instructions for turning this file into a Unix man page. The instructions are something like

nroff -ms -u -rM62 -Tlp /home/kent/avs_dir/mymodule.txt > mymodule.1

for a help file named mymodule.txt created by the **Module Generator**. This will create a file named mymodule.1 or whatever is appropriate for your module's name. Before the Unix man command can access this man page, it must be placed in the Unix man directory tree structure. This is a directory tree located at a commonly accessible place in the Unix file system[1] with the top level name of man. Within this directory is a subdirectory named man1 which should become the home for the mymodule.1 man page and any others that share this **MANPATH**. To find this man page, the path to the top level man page must be specified in one of two ways[2], either by specifying the path using the **MANPATH** environment variable, or by using the -M man option followed by the path. To facilitate the use of man pages, a LIGO wide man directory tree structure needs to be implemented on the LIGO file servers. This should have expanded functionality over what is currently available with the ligobin and ligoman directories[3]. For example, making the source code management software and the shared module library requirements of AVS part of the directory tree, allows the LIGO group to access and share a common version of software.

6 SOURCE CODE MANAGEMENT

Source code management or version control as it is often called is a very important aspect of large software projects. There are several public and commercial systems available for automating many tasks associated with coordinating a team of software developers. The tasks included by most of these systems are maintaining all versions of a program in a recoverable form, prevention

1. See Appendix 5 for the directory tree structure.
2. See the man page for man for details.
3. Found under /home/ufs2 on the LIGO servers.

of simultaneous modification of the same source code, assistance in the merging of two different development tracks into a single version, and automatic documentation of revisions and other changes to log files. These systems establish configuration directory trees from which the source code is checked in and checked out. While the source code is being worked on by one programmer it can be locked against edits by others.

Of course, there are common sense steps that everyone should use to manage their software which don't require the use of these systems. The use of comments to document changes is one that has already been discussed. Maintaining a log file of changes is another simple task. When beginning a software development program, consider allowing for revisions while designing the directory tree for the software. Make the necessary umask protection for the files so that others can read your files but not write to them. This way, the only way someone can check in a change to your code is to come through you.

The LIGO project hasn't settled on a system for source code control, but the basic approaches are the same. The Solaris 2.4 operating system provides one system called SCCS (Source Code Control System) which is documented in the man pages. Another publicly available system is RCS[1] (Revision Control System). The EPICS software uses CVS which sits on top of RCS. There may be a significant advantage to adopting CVS as a standard source code management system for all LIGO software development since it is already being used by the CDS group. There are also commercial products available that add several nice features like graphical interfaces.

APPENDIX 1 AVS EXAMPLE

As a simple example of developing an AVS module, the steps needed to write a random number generator are outlined. There will be no inputs to the module, only parameters. In order for the module to be better integrated in the AVS environment, its output will be either a one, two or three dimensional array of random numbers wrapped up in an AVS field structure. This is important since many supported modules from AVS use the field data structure. In fact, you should consider making all your module inputs and outputs that communicate arrays of data be field data. The list of user specified parameters and their meanings for this module will be the following.

- **method** - integer between 0 and 3 representing which of the random number generation engines ran0, ran1, ran2 or ran3 from the first edition of Numerical Recipes in C are to be used in constructing the array of random numbers. The default will be 3 corresponding to ran3. The widget will be a slider.
- **ndim** - integer of value 1, 2 or 3 representing the dimension of the array of random numbers. The default will be 1 for a one dimensional array. The widget will also be a slider.
- **dimX** - integer between 1 and the largest machine integer representing the dimension size of the X axis. The default will be 1 causing the module to always generate at least one random number. The widget to be used will be the typein_integer.
- **dimY** - integer between 0 and the largest machine integer representing the dimension size of the Y axis. The default will be 0. The widget to be used will be the typein_integer.
- **dimZ** - integer between 0 and the largest machine integer representing the dimension size of the Z axis. The default will be 0. The widget to be used will be the typein_integer.

1. See Chapter 7 in Unix for FORTRAN Programmers by O'Reilly & Associates, Inc.

- **scaling** - float used for scaling the random numbers generated. There are no limits on the value other than it be representable by a floating point number. The default will be 1 causing the random numbers to have a range between 0 and 1. The widget to be used is the typein_real.
- **offset** - float used to offset, by addition, the random numbers after the scaling has been applied. There are no limits on the value other than it be representable by a floating point number. The default will be 0. The widget to be used is the typein_real.
- **seed** - integer used as a seed to the random number generator. The interface to the random number generators requires that the seed be negative, so any integer value less than 0 is allowed. The default value is -71557. The widget to be used is the typein_integer.

There will be a single output port for the array of random numbers stored in a field data structure[1]. This port will be named outPut. Its data type as we said will be field. The dimensions of the field will be unspecified, as will the physical space and the vector length. The data type stored in the field will be float and the spacing will be uniform.

From the main AVS menu, open the network editor. Drag down the **Module Generator** onto the palette. This will setup the parameters for the module in the parameter window to the left of the network editor window. Enter the name of the new module to be created as '**random field**'. Set module type to **Input**, **C** and **Subroutine**. Using the **Edit Parameters** button, enter the eight parameters listed above by renaming Unused 0 through Unused 7 to correspond to the given names in the **Parameter Name** field and also set the data type, the widget type and the range and defaults. For the min and max possible integers, use -2147483647 and 2147483647. After entering all the parameter information, select the **Edit Output Port** button for the **Module Generator** parameter window on the left and rename the Unused 0 port to be outPut. Set its data type to be field and then in the **AVS Field Editor** window set the **Data Type** to be **float** and the **Spacing** to be **uniform**. Leave the other characterizations as **Unspecified**. This completes the designation of the module parameters and ports.

In the **Module Generator** parameter window, specify the directory path to the source code to be whereever you would like to place the source code using the **New Dir** button. This pops up a window for specifying the path. Next select the **New File** button. Again a window will pop up for specifying the file name. Enter *random_field.c* for the name. At this point we can write the AVS module wrapper by selecting the **Write Source** button and also write the AVS help page by selecting the **Write Man** Page button. This will produce the files r*andom_field.c* and *random_field.txt* in the designated directory. At this point you can use the **Edit** button to edit the AVS module wrapper or using a shell and your favorite editor make the edits to put the computational muscle into the code. Once completed the code in *random_field.c* should look like the following where the user's added lines of code are shown in bold.

```
/* mod_gen Version 1                                    */
/* Module Name: "random field" (Input) (Subroutine)     */
/* Author: Kent Blackburn                               */
/* Date Created: Wed Aug 16 09:22:04 1995               */
/*                                                      */
/* This file is automatically generated by the Module Generator (mod_gen)*/
/* Please do not modify or move the contents of this comment block as   */
```

1. See chapter 2 of AVS User's Guide for details

```
/* mod_gen needs it in order to read module sources back in.          */
/*                                                                    */
/* output 0 "outPut" field uniform float                              */
/* param 0 "method" islider 3 0 3                                     */
/* param 1 "ndim" islider 1 1 3                                       */
/* param 2 "dimX" typein_integer 1 0 2147483647                       */
/* param 3 "dimY" typein_integer 0 0 2147483647                       */
/* param 4 "dimZ" typein_integer 0 0 2147483647                       */
/* param 5 "scaling" typein_real 1.00000 FLOAT_UNBOUND FLOAT_UNBOUND  */
/* param 6 "offset" typein_real 0.00000 FLOAT_UNBOUND FLOAT_UNBOUND   */
/* param 7 "seed" typein_integer -71557 -2147483647 -1                */
/* End of Module Description Comments                                 */

#include <stdio.h>
#include <avs/avs.h>
#include <avs/port.h>
#include <avs/field.h>

/* ----> START OF USER-SUPPLIED CODE SECTION #1 (INCLUDE FILES, GLOBAL VARIABLES)*/
/* LIGO HEADER KEYWORDS */
/* program name: random_field
/* application: AVS module */
/* author: Dr. James Kent Blackburn */
/* organization: LIGO */
/* project: end-to-end model */
/* hardware developed on: Sun Sparcstation */
/* compiler developed on: C Development Set  (CDS) SPARCompilers 2.0.1 */
/* purpose: generate AVS 1D, 2D or 3D random number fields */
/* reference documents: AVS Software Style Guide */
/* modification history: JKB | 8/15/95 | fixed bug in NRC code that assumed
                  rand() system function used 32 bit integers, it
                  actually uses 16 bit integers on the Sun */
#include <stdlib.h>
#include <math.h>
#include "random_field.h"
/* <---- END OF USER-SUPPLIED CODE SECTION #1                          */

/* ***************************************/
/*  Module Description                   */
/* ***************************************/
int random_field_desc()
{

int in_port, out_port, param, iresult;
extern int random_field_compute();

AVSset_module_name("random field", MODULE_DATA);

/* Output Port Specifications            */
out_port = AVScreate_output_port("outPut", "field uniform float");

/* Parameter Specifications              */
param = AVSadd_parameter("method", "integer", 3, 0, 3);
AVSconnect_widget(param, "islider");
param = AVSadd_parameter("ndim", "integer", 1, 1, 3);
AVSconnect_widget(param, "islider");
param = AVSadd_parameter("dimX", "integer", 1, 0, 2147483647);
AVSconnect_widget(param, "typein_integer");
param = AVSadd_parameter("dimY", "integer", 0, 0, 2147483647);
AVSconnect_widget(param, "typein_integer");
param = AVSadd_parameter("dimZ", "integer", 0, 0, 2147483647);
AVSconnect_widget(param, "typein_integer");
param = AVSadd_float_parameter("scaling", 1.00000, FLOAT_UNBOUND,
FLOAT_UNBOUND);
AVSconnect_widget(param, "typein_real");
param = AVSadd_float_parameter("offset", 0.00000, FLOAT_UNBOUND,
FLOAT_UNBOUND);
AVSconnect_widget(param, "typein_real");
param = AVSadd_parameter("seed", "integer", -71557, -2147483647, -1);
AVSconnect_widget(param, "typein_integer");

AVSset_compute_proc(random_field_compute);
```

```
/* ----> START OF USER-SUPPLIED CODE SECTION #2 (ADDITIONAL SPECIFICATION INFO)*/
/* <---- END OF USER-SUPPLIED CODE SECTION #2                    */
return(1);
}

/* ****************************************/
/* Module Compute Routine                 */
/* ****************************************/
int random_field_compute( outPut, method, ndim, dimX, dimY, dimZ,
scaling, offset, seed)
AVSfield_float **outPut;
int method;
int ndim;
int dimX;
int dimY;
int dimZ;
float *scaling;
float *offset;
int seed;
{

/* ----> START OF USER-SUPPLIED CODE SECTION #3 (COMPUTE ROUTINE BODY)   */

 int i, npoints;
 int dims1[1], dims2[2], dims3[3];
 float *fptr, data_min, data_max;
 float ran0(), ran1(), ran2(), ran3();

 data_min = *offset;
 data_max = *scaling + *offset;
/* If memory for outPut port already exists, then free it up to avoid using all the memory */
 if (*outPut) AVSfield_free(*outPut);

 /* create the appropriate field depending on the dimension */
 switch (ndim)
     {
     case 1:
          dims1[0] = dimX;
          npoints = dimX;
/* Allocate the memory needed for a 1D  outPut port here
          *outPut = (AVSfield_float *)
          AVSdata_alloc("field 1D 1-space scalar uniform float", dims1);
          break;
     case 2:
          dims2[0] = dimX;
          dims2[1] = dimY;
          npoints = dimX * dimY;
/* Allocate the memory needed for a 2D  outPut port here
          *outPut = (AVSfield_float *)
          AVSdata_alloc("field 2D 2-space scalar uniform float", dims2);
          break;
     case 3:
          dims3[0] = dimX;
          dims3[1] = dimY;
          dims3[2] = dimZ;
          npoints = dimX * dimY * dimZ;
/* Allocate the memory needed for a 3D  outPut port here
          *outPut = (AVSfield_float *)
          AVSdata_alloc("field 3D 3-space scalar uniform float", dims3);
          break;
     }
 if (*outPut == NULL)
   {
   AVSerror("Allocation of outPut field failed.");
/* Note that returning azero instead of a one implies an error to AVS */
   return(0);
   }
 fptr = (float *) (*outPut)->data;
 (*outPut)->minimum = &data_min;
 (*outPut)->maximum = &data_max;

 /* use the random number function associated with the method */
```

```
switch (method)
    {
    case 0:
        for (i=0; i<npoints; i++) fptr[i] = ran0(&seed);
        break;
    case 1:
        for (i=0; i<npoints; i++) fptr[i] = ran1(&seed);
        break;
    case 2:
        for (i=0; i<npoints; i++) fptr[i] = ran2(&seed);
        break;
    case 3:
        for (i=0; i<npoints; i++) fptr[i] = ran3(&seed);
        break;
    }

/* rescale the random numbers if the *scaling != 1 */
if (*scaling != 1)
  for (i=0; i<npoints; i++) fptr[i] *= *scaling;

/* offset the random numbers if the *offset != 0 */
if (*offset != 0)
  for (i=0; i<npoints; i++) fptr[i] += *offset;

/* <---- END OF USER-SUPPLIED CODE SECTION #3                    */
return(1);
}

/* ************************************************************************/
/* Initialization for modules contained in this file.          */
/* ************************************************************************/
static int ((*mod_list[])()) = {
random_field_desc
};
#define NMODS (sizeof(mod_list) / sizeof(char *))

AVSinit_modules()
{
AVSinit_from_module_list(mod_list, NMODS);
}

/* ----> START OF USER-SUPPLIED CODE SECTION #4 (SUBROUTINES, FUNCTIONS, UTILITY ROU-
TINES)*/

/* Random Number Generator 0: From Numerical Recipes in C - Chapter 7 */

float ran0(idum)
int *idum;
{
 static float y, maxran, v[98];
 float dum;
 static int iff=0;
 int j;
 unsigned short i,k;

 if (*idum < 0 || iff == 0)
  {
   iff = 1;
   i = 2;
   do {
      k = i;
      i = i << 1;
     } while (i);
   maxran = k;
   SRAND(*idum);
   *idum = 1;
   for (j=1; j<=97; j++) dum = RAND();
   for (j=1; j<=97; j++) v[j] = RAND();
   y = RAND();
  }
 j = 1 + 97.0 * y / maxran;
 if (j > 97 || j < 1) AVSerror("ran0 error 1: See random_field.c");
```

```
 y = v[j];
 v[j] = RAND0;
 return y / maxran;
}

/* Random Number Generator 1: From Numerical Recipes in C - Chapter 7 */

float ran1(idum)
int *idum;
{
 static long ix1, ix2, ix3;
 static float r[98];
 float temp;
 static int iff=0;
 int j;

 if (*idum < 0 || iff == 0)
  {
   iff = 1;
   ix1 = (IC1 - (*idum)) % M1;
   ix1 = (IA1 * ix1 + IC1) % M1;
   ix2 = ix1 % M2;
   ix1 = (IA1 * ix1 +IC1) % M1;
   ix3 = ix1 % M3;
   for (j=1; j<=97; j++)
     {
      ix1 = (IA1 * ix1 + IC1) % M1;
      ix2 = (IA2 * ix2 + IC2) % M2;
      r[j] = (ix1 + ix2 * RM2) * RM1;
     }
   *idum = 1;
  }
 ix1 = (IA1 * ix1 + IC1) % M1;
 ix2 = (IA2 * ix2 + IC2) % M2;
 ix3 = (IA3 * ix3 + IC3) % M3;
 j = 1 + ((97 * ix3) / M3);
 if (j > 97 || j < 1) AVSerror("ran1 error 1: See random_field.c");
 temp = r[j];
 r[j] = (ix1 + ix2 * RM2) * RM1;
 return temp;
}

/* Random Number Generator 2: From Numerical Recipes in C - Chapter 7 */

float ran2(idum)
int *idum;
{
 static long iy, ir[98];
 static int iff = 0;
 int j;

 if (*idum < 0 || iff == 0)
  {
   iff = 1;
   if ((*idum = (IC - (*idum)) % M) < 0) *idum = -(*idum);
   for (j=1; j<=97; j++)
     {
      *idum = (IA * (*idum) + IC) % M;
      ir[j] = (*idum);
     }
   *idum = (IA * (*idum) + IC) % M;
   iy = (*idum);
  }
 j = 1 + 97.0 * iy / M;
 if (j > 97 || j < 1) AVSerror("ran2 error 1: See random_field.c");
 iy = ir[j];
 *idum = (IA * (*idum) + IC) % M;
 ir[j] = (*idum);
 return (float) iy / M;
}

/* Random Number Generator 3: From Numerical Recipes in C - Chapter 7 */
```

```
float ran3(idum)
int *idum;
{
 static int inext, inextp;
 static long ma[56];
 static int iff = 0;
 long mj, mk;
 int i, ii, k;

 if (*idum < 0 || iff == 0)
  {
   iff = 1;
   mj = MSEED - (*idum < 0 ? -(*idum): (*idum));
   mj %= MBIG;
   ma[55] = mj;
   mk = 1;
   for (i=1; i<=54; i++)
     {
      ii = (21 * i) % 55;
      ma[ii] = mk;
      mk = mj - mk;
      if (mk < MZ) mk += MBIG;
      mj = ma[ii];
     }
   for (k=1; k<=4; k++)
     for (i=1; i<=55; i++)
       {
        ma[i] -= ma[1+(i+30) % 55];
        if (ma[i] < MZ) ma[i] += MBIG;
       }
   inext = 0;
   inextp = 31;
   *idum = 1;
  }
 if (++inext == 56) inext = 1;
 if (++inextp == 56) inextp = 1;
 mj = ma[inext] - ma[inextp];
 if (mj < MZ) mj += MBIG;
 ma[inext] = mj;
 return mj*FAC;
}

/* <---- END OF USER-SUPPLIED CODE SECTION #4                    */
```

The random_field.c file includes a C header file named random_field.h that the developer must also supply. It contains the following set of macros needed by the random number generation software engines.

```
#ifndef __RANDOM_FIELD__

#define __RANDOM_FIELD__

/* Macros used by ran0 method */

#define  RAND  rand
#define  SRAND srand

/* Macros used by ran1 method */

#define  M1   259200
#define  IA1  7141
#define  IC1  54773
#define  RM1  (1.0 / M1)
#define  M2   134456
#define  IA2  8121
```

```
#define  IC2  28411
#define  RM2  (1.0 / M2)
#define  M3   243000
#define  IA3  4561
#define  IC3  51349

/* Macros used by ran2 method */

#define  M    714025
#define  IA   1366
#define  IC   150889

/* Macros used by ran3 method */

#define  MBIG  1000000000
#define  MSEED 161803398
#define  MZ   0
#define  FAC  (1.0 / MBIG)

#endif
```

The *#ifndef __RANDOM_FIELD__, #define __RANDOM_FIELD__, ... , #endif* surrounding the macros in the header are important for avoiding the reading of macros twice in the event that this header file was indirectly included a second time.

The same module could have been written in FORTRAN by simply replacing the step selecting the AVS module as C with FORTRAN. This results in a FORTRAN wrapper being generated once the file name is specified as *random_field.f* and the **Write Source** button is selected. In FORTRAN the field structure must be accessed through AVS library routines since standard FORTRAN doesn't support structures. Also, the data in the field structure must be addressed using offsets from an arbitrary array. Other than these differences the FORTRAN code looks very similar to the C version. Notice however, that there is no USER-SUPPLIED CODE SECTION # 1 for the FORTRAN wrapper, so the LIGO HEADER KEYWORDS have been placed in SECTION # 2. Here is the equivalent FORTRAN **random field** module source code.

```
C mod_gen Version 1
C Module Name: "random field" (Input) (Subroutine)
C Author: Kent Blackburn
C Date Created: Wed Aug 23 11:03:30 1995
C
C This file is automatically generated by the Module Generator (mod_gen)
C Please do not modify or move the contents of this comment block as
C mod_gen needs it in order to read module sources back in.
C
C output 0 "outPut" field uniform float
C param 0 "method" islider 3 0 3
C param 1 "ndim" islider 1 1 3
C param 2 "dimX" typein_integer 1 0 2147483647
C param 3 "dimY" typein_integer 0 0 2147483647
C param 4 "dimZ" typein_integer 0 0 2147483647
C param 5 "scaling" typein_real 1.00000 FLOAT_UNBOUND FLOAT_UNBOUND
C param 6 "offset" typein_real 0.00000 FLOAT_UNBOUND FLOAT_UNBOUND
C param 7 "seed" typein_integer -71557 -2147483647 -1
C End of Module Description Comments


C ****************************************
C  Module Description
C ****************************************
      integer function random_field_desc()
      implicit none
      include 'avs/avs.inc'
```

```
      integer in_port, out_port, param, iresult
      external random_field_compute
      integer random_field_compute

      call AVSset_module_name('random field', 'data')
      call AVSset_module_flags(single_arg_data)

COutput Port Specifications
      out_port = AVScreate_output_port('outPut', 'field uniform float')

CParameter Specifications
      param = AVSadd_parameter('method', 'integer', 3, 0, 3)
      call AVSconnect_widget(param, 'islider')
      param = AVSadd_parameter('ndim', 'integer', 1, 1, 3)
      call AVSconnect_widget(param, 'islider')
      param = AVSadd_parameter('dimX', 'integer', 1, 0, 2147483647)
      call AVSconnect_widget(param, 'typein_integer')
      param = AVSadd_parameter('dimY', 'integer', 0, 0, 2147483647)
      call AVSconnect_widget(param, 'typein_integer')
      param = AVSadd_parameter('dimZ', 'integer', 0, 0, 2147483647)
      call AVSconnect_widget(param, 'typein_integer')
      param = AVSadd_parameter('scaling','real', 1.00000,
     $ FLOAT_UNBOUND, FLOAT_UNBOUND)
      call AVSconnect_widget(param, 'typein_real')
      param = AVSadd_parameter('offset','real', 0.00000,
     $ FLOAT_UNBOUND, FLOAT_UNBOUND)
      call AVSconnect_widget(param, 'typein_real')
      param = AVSadd_parameter('seed', 'integer', -71557, -2147483647,
     $ -1)
      call AVSconnect_widget(param, 'typein_integer')

      call AVSset_compute_proc(random_field_compute)
C ----> START OF USER-SUPPLIED CODE SECTION #2 (ADDITIONAL SPECIFICATION INFO)

C LIGO HEADER KEYWORDS
C program name: random_field
C application: AVS module
C author: Dr. James Kent Blackburn
C organization: LIGO
C project: end-to-end model
C hardware developed on: Sun Sparcstation
C compiler developed on: Sun Fortran version 2.0.1
C purpose: generate AVS 1D, 2D or 3D random number fields
C reference documents: AVS Software Standards Guide
C modification history:

C <---- END OF USER-SUPPLIED CODE SECTION #2
      random_field_desc = 1
      return
      end


C ****************************************
C Module Compute Routine
C ****************************************
      integer function random_field_compute( outPut, method, ndim,
     $ dimX, dimY, dimZ, scaling, offset, seed)
      implicit none
      include 'avs/avs.inc'
      integer outPut
      integer method
      integer ndim
      integer dimX
      integer dimY
      integer dimZ
      real scaling
      real offset
      integer seed

C ----> START OF USER-SUPPLIED CODE SECTION #3 (COMPUTE ROUTINE BODY)

      integer npoints, iresult, offptr
```

```
      integer dim1(1), dim2(2), dim3(3)
      real datamin(1), datamax(1), datafld(1)
      datamin(1) = offset
      datamax(1) = scaling + offset
C     If memory for outPut port already exists, then free it up to avoid using all the memory
      if (outPut .ne. 0) call AVSfield_free(outPut)
      if (ndim .eq. 1) then
       dim1(1) = dimX
       npoints = dimX
C     Allocate the memory needed for a 1D outPut port
       outPut =
     &  AVSdata_alloc('field 1D 1-space scalar uniform float',dim1)
      else if (ndim .eq. 2) then
       dim2(1) = dimX
       dim2(2) = dimY
       npoints = dimX * dimY
C     Allocate the memory needed for a 2D outPut port
       outPut =
     &  AVSdata_alloc('field 2D 2-space scalar uniform float',dim2)
      else
       dim3(1) = dimX
       dim3(2) = dimY
       dim3(3) = dimZ
       npoints = dimX * dimY * dimZ
C     Allocate the memory needed for a 3D outPut port
       outPut =
     &  AVSdata_alloc('field 3D 3-space scalar uniform float',dim3)
      endif
      if (outPut .eq. 0) then
       call AVSerror('Allocation of outPut field failed.')
       random_field_compute = 0
       return
      endif
      call AVSfield_set_minmax(outPut, datamin, datamax)
C get the offset 'offptr' from datafld to the field's data array
      iresult = AVSfield_data_offset(outPut, datafld, offptr)
      if (iresult .eq. 0) then
       call AVSerror('Unable to get offset to outPut data.')
       random_field_compute = 0
       return
      endif
C dereference the pointer to the fields data array by passing down
      call randomize_data(datafld(offptr+1),npoints, method, seed)

C <---- END OF USER-SUPPLIED CODE SECTION #3
      random_field_compute = 1
      return
      end


C ************************************************************************
C Initialization for modules contained in this file.
C ************************************************************************
      subroutine AVSinit_modules
      include 'avs/avs.inc'

      external random_field_desc
      integer random_field_desc
      call AVSmodule_from_desc(random_field_desc)
      end


C ----> START OF USER-SUPPLIED CODE SECTION #4 (SUBROUTINES, FUNCTIONS, UTILITY ROUTINES)

C This routine populates the data array of the outPut field with
C random numbers. It is called to demonstrate using the offset
C from the AVSfield_data_offset() routine as a dereferencing trick

      subroutine randomize_data(array, npoints, method, seed)
      real array(1)
      integer npoints, method, seed
      real ran0, ran1, ran2, ran3
```

```
      if (method .eq. 0) then
       do 10 i = 1, npoints
        array(i) = ran0(seed)
10     continue
      else if (method .eq. 1) then
       do 11 i = 1, npoints
        array(i) = ran1(seed)
11     continue
      else if (method .eq. 2) then
       do 12 i = 1, npoints
        array(i) = ran2(seed)
12     continue
      else
       do 13 i = 1, npoints
        array(i) = ran3(seed)
13     continue
      endif
      return
      end
```

C Random Number Generator 0: from Numerical Recipes - Chapter 7

```
      function ran0(idum)
      dimension v(97)
      data iff /0/
      if (idum .lt. 0 .or. iff .eq. 0) then
       iff = 1
       iseed = abs(idum)
       idum = 0
       dum = rand(iseed)
       do 11 j = 1, 97
        dum = rand(0)
11     continue
       do 12 j = 1, 97
        v(j) = rand(0)
12     continue
       y = rand(0)
      endif
      j = 1 + int(97 * y)
      if (j .gt. 97 .or. j .lt. 1) then
       call AVSerror('ran0 error 1: See random_field.f')
      endif
      y = v(j)
      ran0 = y
      v(j) = rand(0)
      return
      end
```

C Random Number Generator 1: from Numerical Recipes - Chapter 7

```
      function ran1(idum)
      dimension r(97)
      parameter (m1=259200, ia1=7141, ic1=54773, rm1=1./m1)
      parameter (m2=134456, ia2=8121, ic2=28411, rm2=1./m2)
      parameter (m3=243000, ia3=4561, ic3=51349)
      data iff /0/
      if (idum .lt. 0 .or. iff .eq. 0) then
       iff = 1
       ix1 = mod(ic1 - idum, m1)
       ix1 = mod(ia1 * ix1 + ic1, m1)
       ix2 = mod(ix1, m2)
       ix1 = mod(ia1 * ix1 + ic1, m1)
       ix3 = mod(ix1, m3)
       do 11 j = 1, 97
        ix1 = mod(ia1 * ix1 + ic1, m1)
        ix2 = mod(ia2 * ix2 + ic2, m2)
        r(j) = ((ix1) + (ix2) * rm2) * rm1
11     continue
       idum = 1
```

```
      endif
      ix1 = mod(ia1 * ix1 + ic1, m1)
      ix2 = mod(ia2 * ix2 + ic2, m2)
      ix3 = mod(ia3 * ix3 + ic3, m3)
      j = 1 + (97 * abs(ix3)) / m3
      if (j .gt. 97 .or. j .lt. 1) then
       print*,j
       call AVSerror('ran1 error 1: See random_field.f')
      endif
      ran1 = r(j)
      r(j) = ((ix1) + (ix2) * rm2) * rm1
      return
      end
```

C Random Number Generator 2: from Numerical Recipes - Chapter 7

```
      function ran2(idum)
      dimension ir(97)
      parameter (m=714025, ia=1366, ic=150889, rm=1./m)
      data iff /0/
      if (idum .lt. 0 .or. iff .eq. 0) then
       iff = 1
       idum = mod(ic - idum, m)
       do 11 j = 1, 97
        idum = mod(ia * idum +ic, m)
        ir(j) = idum
 11    continue
       idum = mod(ia * idum + ic, m)
       iy = idum
      endif
      j = 1 + (97 * iy) / m
      if (j .gt. 97 .or. j .lt. 1) then
       call AVSerror('ran2 error 1: See random_field.f')
      endif
      iy = ir(j)
      ran2 = iy * rm
      idum = mod(ia * idum + ic, m)
      ir(j) = idum
      return
      end
```

C Random Number Generator 3: from Numerical Recipes - Chapter 7

```
      function ran3(idum)
      parameter (mbig=1000000000, mseed=161803398, mz=0, fac=1./mbig)
      dimension ma(55)
      data iff /0/
      if (idum .lt. 0 .or. iff .eq. 0) then
       iff = 1
       mj = mseed - iabs(idum)
       mj = mod(mj, mbig)
       ma(55) = mj
       mk = 1
       do 11 i = 1, 54
        ii = mod(21 * i, 55)
        ma(ii) = mk
        mk = mj - mk
        if (mk .lt. mz) mk = mk + mbig
        mj = ma(ii)
 11    continue
       do 13 k = 1, 4
        do 12 i = 1, 55
         ma(i) = ma(i) - ma(1 + mod(i + 30, 55))
         if (ma(i) .lt. mz) ma(i) = ma(i) + mbig
 12      continue
 13    continue
       inext = 0
       inextp = 31
       idum = 1
      endif
```

```
inext = inext + 1
if (inext .eq. 56) inext = 1
inextp = inextp + 1
if (inextp .eq. 56) inextp = 1
mj = ma(inext) - ma(inextp)
if (mj .lt. mz) mj = mj + mbig
ma(inext) = mj
ran3 = mj * fac
return
end

C <---- END OF USER-SUPPLIED CODE SECTION #4
```

The AVS help page is written to the file *random_field.txt*. It is a template for the on-line help available through AVS. Initially it looks like the following.

NAME

random field - PUT A BRIEF DESCRIPTION HERE

SUMMARY

Name	random field	
Type	Input	
Inputs	NONE	
Outputs	outPut -	field uniform float

Parameters

Name		Type	Default	Choices
method	islider	3	0	3
ndim	islider	1	1	3
dimX	typein_integer	1	0	2147483647
dimY	typein_integer	0	0	2147483647
dimZ	typein_integer	0	0	2147483647
scaling	typein_real	1.00	0.00	0.00
offset	typein_real	0.00	0.00	0.00
seed	typein_integer	-71557	-2147483647	-1

DESCRIPTION

INPUTS

PARAMETERS

method (islider)

ndim (islider)

dimX (typein_integer)

dimY (typein_integer)

dimZ (typein_integer)

scaling (typein_real)

offset (typein_real)

seed (typein_integer)

OUTPUTS

outPut - field uniform float

EXAMPLE NETWORKS

RELATED MODULES

SEE ALSO

After the user has filled in the various sections, a more complete help page for this **random field** module would look something like the following

NAME

random field - generates a 1D, 2D or 3D field of random numbers having floating point values between scaling+offset and offset.

SUMMARY

Name	random field	
Type	Input	
Inputs	NONE	
Outputs	outPut -	field uniform float

Parameters

Name		Type	Default	Choices
method	islider	3	0	3
ndim	islider	1	1	3
dimX	typein_integer	1	0	2147483647
dimY	typein_integer	0	0	2147483647
dimZ	typein_integer	0	0	2147483647
scaling	typein_real	1.00	0.00	0.00
offset	typein_real	0.00	0.00	0.00
seed	typein_integer	-71557	-2147483647	-1

DESCRIPTION

This AVS module generates an array of random numbers of dimension 1D, 2D or 3D depending on the ndim parameter. The size of the array is determined by the (dimX,dimY,dimZ) parameters. By default the random numbers will be floats between 0 and 1, but using the scaling and offset parameters these can be set to be any numbers specified by scaling * RN + offset where offset is a random number from 0 to 1. The user can also specify a seed to be used for starting the random number generator. The function used to generate the random number series is given by the value of the method parameter. All functions are based on code from the first edition of Numerical Recipes in C.

INPUTS

None

PARAMETERS

method (islider)

Integer between 0 and 3 representing which of the random number generation engines ran0, ran1, ran2, ran3 from the first edition of Numerical Recipes in C are to be used in constructing the array of random numbers. The default with be 3 corresponding to ran3.

ran0 is just a randomization of random numbers generated by the rand() function available with the system.

ran1 uses three linear congruential generators. One generates the most significant part of a random number, the second generates the least significant part of a random number and the third is used by the suffling routine. Its period is for all practical purposes infinite.

ran2 is a much faster routine at the expense of providing less discreteness in the random numbers generated.

ran3 is based on the subtractive method instead of the linear conguential method of ran1 and ran2.

ndim (islider)

Integer of value 1, 2, or 3 representing the dimension of the

array of random numbers. The default will be 1 for a one dimensional array.

dimX (typein_integer)

Integer between 1 and the largest machine integer representing the dimension size of the X axis. The default will be 1 causing the module to always generate at least one random number.

dimY (typein_integer)

Integer between 0 and the largest machine integer representing the dimension size of the Y axis. The default will be 0.

dimZ (typein_integer)

Integer between 1 and the largest machine integer representing the dimension size of the Z axis. The default will be 0.

scaling (typein_real)

Float used for scaling the random numbers generated. There are no limits on the value other than it be representable by a floating point number. The default will be 1.0 causing the random numbers to have a range between 0 and 1.

offset (typein_real)

Float used to offset, by addition, the random numbers after the scaling has been applied. There are no limits on the value other than it be representable by a floating point number. The default will be 0.0.

seed (typein_integer)

Integer used as a seed to the random number generator. The interface to the random number generators requires that the seed be negative, so any integer value less than 0 is allowed. The default value is -71557.

OUTPUTS

outPut - field uniform float containing the 1D, 2D or 3D array of random numbers.

EXAMPLE NETWORKS

The output from random field is a standard AVS field output and can be used in conjunction with all AVS field modules. A simple example is to feed the output of random field into the AVSGraph module and test the various parameters for a 1D field.

RELATED MODULES

Any AVS field module.

SEE ALSO

AVS Developer's Guide

As discussed in Section 5.4 on Unix Man Pages, setting the **AVS_MG_TROFF** environment variable prior to starting AVS causes the Write Man Page button to generate a troff formatted file that can then be turned into a standard Unix man page and made available to all users on the system.

At present the files from this example are located in ~kent/avs_dir/modules/random_field on the LIGO file server. Once the server has adopted a standard directory format, these will be moved.

In this examples, the AVSfield_float structure that is used to hold the random numbers is given by:

```
typedef struct {
        int ndim;               /* no. of computational dimensions */
        int nspace;             /* no. of coordinate dimentions */
        int veclen;             /* no. of values per data element */
        int type;               /* data type */
        int size;               /* size of each value in data element */
        int single_block;       /* internal, type of memory allocation */
        int uniform;            /* mapping type: Uniform, Rectilinear, or Irreg */
        int flags;              /* data validity flags */
        int *dimensions;        /* demension along each axis; length is ndim */
        float *points;          /* coordinates for fields */
        float *data;            /* the field data itself as floats */
        float *min_extent;      /* range of data; array size is nspace */
        float *max_extent;      /* range of data; array size is nspace */
        float *minimum;         /* min data values for each value in a data element */
        float *maximum;         /* max data values for each value in a data element */
        int shm_key;            /* internal, shared memory key */
        int shm_id;             /* internal, shared memory identifier */
        char *shm_base;         /* internal, shared memory base address */
        char *units;            /* units for each component */
        int shm_size;           /* internal, shared memory segment size */
        int mesh_id;            /* unique id for the "points" information */
    } AVSfield_float;
```

APPENDIX 2 OOP OVERVIEW

In a single statement, object oriented programming means to organize software as a collection of discrete objects that contain both data structure and functionality. This is in sharp contrast to the procedural programming approach of having data structures and functionality as distinctly separate identities that are only loosely coupled. The exact characteristics of an object oriented programming language are still being debated. However, they universally are accepted to include the characteristics of identity, classification, polymorphism and inheritance.

Identity is characterized by discrete, distinguishable units called objects. These objects are apparent from the data structures of the real world, for example this document can be thought of as an object, being both discrete and distinguishable from other documents.

Classification simply means that objects of identical data attributes and functionality or operations get grouped together in a class. Thus all documents similar to this document might be grouped into a Documents class. The class is an abstraction that describes the properties that characterize objects. Each object is said to be an instance of its class.

Polymorphism means that a particular operations which is conceptually in common to different classes may behave differently for the different classes. For example a *moveto* operation would behave differently for a document than it would for a graph and in fact may be parameterized differently. The operation implemented by a class is usually called a method.

Inheritance is the sharing of attributes and methods among classes in a hierarchical way. This allows classes to be defined with very broad properties and be refined into successively finer subclasses. Each subclass is said to inherit all the properties of its superclass while added the specialized properties that result from the refinements. This ability to inherit properties greatly reduces

repetition in designs and software which is considered one of the strongest reasons for using object oriented programming.

APPENDIX 3 RANDOM NUMBER GENERATORS

System supplied random number generators are a source of endless frustration if they are ever trusted to provide significantly long list of values or values without large gaps in their range. Most can only produce sequences of about 32K numbers before the sequence begins to repeat. Even the man pages for supplied C and FORTRAN random number generators on the Sun Solaris platforms worn of their limitations. Serious users of random number generators should always implement in their codes the option to use more than one method for generating random numbers to guarantee that the statistical properties of results from using random numbers generated by an algorithm are not influenced by short periods or large gaps in the sequence. The random field module found in Appendix 1 is an example of the kind of flexibility that should go into programs using random number generators.

To avoid having every software developer using different random number generators, it is everyone's best interest to adopt a standard LIGO random number generator and make it a part of the standard LIGO software library. Statistical tests on the random numbers generated must included in the normal documentation for software.

APPENDIX 4 HEADER KEYWORDS

Here is a list of standard keywords that should be adopted for all LIGO AVS module source codes and all other simulation software associated LIGO. Since source code changes are lost when modifications are made to any AVS module interfaces, these header keywords must appear in one of the USR-SUPPLIED CODE SECTIONS. It is recommended that SECTION # 1 (SECTION # 2 for FORTRAN wrappers) be designated the appropriate location.

- program name:
- application
- author:
- organization:
- project:
- hardware originally developed on:
- compiler originally developed on:
- purpose:
- reference documents:
- modification history:

Most of these keywords will also appear in any subroutines or functions with a couple of slight variations.

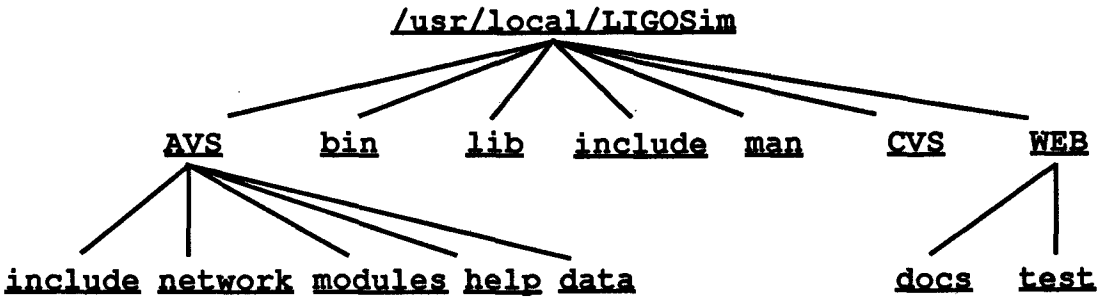
- routine name:
- application:
- author:
- organization:
- project:

- hardware originally developed on:
- compiler originally developed on:
- purpose:
- reference documents:
- modification history:

Of course developers can extend this list to include many other keywords. Often headers will include descriptions of variables and called functions. Descriptions of variables used to represent formulae are often quite valuable when the formulae contains characters not represented by the ASCII characters such as greek letters and probably should be included when needed.

APPENDIX 5 DIRECTORY TREE

The following is a template for a directory tree on a unix file server to be shared by all workstations in the LIGO project. This template includes branches for executable, libraries, headers files, man pages, AVS networks, AVS modules, AVS help, WEB[1] pages for documents and a WEB test area for verification that users have successfully generated HTML files. And an area for the source code management system shown here as CVS.



Assuming this directory structure, there are several useful setup option that should be added to your **$HOME/.avsrc** file. These will allow your AVS session to locate the AVS network files, AVS data files and the AVS help files. Edit your **$HOME/.avsrc** file and place the following lines in it if they are not already represented.

```
HelpPath $AVS_PATH/runtime/help:/usr/local/LIGOSim/AVS/help
NetworkDirectory /usr/local/LIGOSim/AVS/network
DataDirectory /usr/local/LIGOSim/AVS/data
```

This should provide the needed links to this directory tree from within AVS. A useful setup option for personalizing the AVS modules available is the ModuleLibraries option which is set to a file that holds information about all modules that are available in a particular module library.

1. The WEB/docs directory is mirrored to the LIGO WWW directory.