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

Document Type	**LIGO-T040035-00-Z**	January 23, 2004

MATLAB Coding Guidelines

Keith Thorne

Distribution of this draft:

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

Processed with LaTeX on 2004/03/03

Contents

1 Introduction

The Penn State University Relativity Group (PSURG), part of the Center for Gravitational Wave Physics, is now actively developing software for the analysis of LIGO data. It has been decided to use MATLAB (a product of The MathWorks, Inc.) for the development of software for data analysis. As this is a collaborative effort, it behooves us to have rules for organizing and creating our MATLAB-based software.

In addition, it is anticipated that substantial parts of our code may be subject to external use and review. To that end, we should follow guidelines in coding and organization which ensure that our software is done in a professional manner, reflective of all the hard work which has gone into creating it.

At present, the LIGO Collaboration has a standard which applies to the LIGO/LSC Algorithm Library (LAL). The LAL software is written in the C programming language. This existing standard is not easily adapted to the particular syntax of MATLAB, nor to software development outside the LAL environment.

The following guidelines attempt to address the need for rules and guidelines for producing software for data analysis using the MATLAB language. It is derived from ideas in the LAL standard, some MATLAB-specific standards and personal experience in using and developing coding standards for commercial projects.

2 Goals

The goals of these guidelines are:

- Provide rules and guidelines which encourage good software development practices
- Ensure sufficient documentation to allow non-originators to maintain MATLAB code.
- Create achievable goals which strike a balance between requirements and time constraints
- Address the handling and integration of legacy code
- Create uniform methods for coding and documentation to support reviews of the code

3 Files and Organization

3.1 Packages

A package is a set of related functions which carry out some well-defined data analysis task.

3.1.1 All code files for a package shall have the same parent directory

The parent directory shall share the name of the package (i.e. FrameIO, BlockNormal). This parent directory may have one or more subdirectories. The goal is to allow the entire package to be found starting from this parent directory.

3.1.2 Packages should have a test suite

When designing and implementing a package, it is required that a set of test functions will be provided which test the critical operations of the package. The best test suite would have a separate test for each function. These tests can be held in a 'test' sub-directory under the parent directory.

3.1.3 Packages should not duplicate functions in other packages

Within a project (made up of several packages), a given MATLAB function should only occur once. While this leads to packages being dependent on other packages, it prevents proliferation of incompatible version of the same function (and no fair changing your version by a single letter to keep it).

3.1.4 Functions with project-wide use should be placed in a Utilities package

Every project should have a Utilities package to hold those functions which are used by several packages in the project, but whose function is not uniquely associated with any one package.

3.1.5 Binary files should not be stored in the revision control system

Revision control systems, such as CVS, are designed to handle text-based files, as they store only the changes between one revision and the next. If binary files are stored, the revision control system often has to store the whole file for each revision. Also, storing files which are built from other files (executables, libraries, etc.) makes it easy for them to become out of sync with the source code. Building of executables and libraries should be handled outside revision control. However, the scripts and make files to build the executables and libraries should be under revision control.

3.1.6 Packages should have a README file

This README file (in the parent directory) should describe the main components and functions of the package, how to build any required executables and libraries, the location of test suites, and (if possible) a listing of all files in the packages.

3.2 Files

3.2.1 When possible, re-use existing functions

Developing a function that is correct, readable and reasonably flexible can be a significant task. It may be quicker or surer to find an existing function that provides some or all of the required functionality. It also help maintainability.

3.2.2 A large code block appearing in multiple files should re-packaged as a function

It is much easier to manage changes if code appears in only one file.

3.2.3 Locally-used sub-functions can be placed in the main function file

A sub-function used only by another function may be packaged as its sub-function in the same file. This should only be done if it makes the code easier to read and maintain.

3.3 Input and Output

3.3.1 Input and output should be in functions separate from computation

Output requirements are subject to change without notice. Input format and content are subject to change and are often messy. Localizing the code that deals with them improves maintainability. Avoid mixing input or output code with computation, except for pre-processing, in a single function. Mixed-purpose functions are unlikely to be reusable.

3.3.2 Output should be formatted for ease of use

If the output will most likely be read by a human, make it self-descriptive and easy to read. If the output is more likely to be read by software than a person, make it easy to parse. If both are important, make the output easy to parse and write a formatter function to produce a human-readable version.

4 Naming Conventions

4.1 Variables

The names of variables should document their meaning and use.

4.1.1 Variable names should be in mixed case, starting with lowercase

```
sigmoid, detectorSize, heightOfGraph
```

This is common practice in the C++ development community. MathWorks sometimes starts variables with uppercase, but that usage is commonly reserved for objects and structures in other languages.

4.1.2 Only scratch variables may have very short names of limited meaning

In practice, most variables should have meaningful names. The use of very short names should be reserved for conditions where they clarify the structure of the statements. Scratch variables used for temporary storage or indices can be kept short. A programmer reading such variables should be able to assume that their value is not used outside a few lines of code. Common scratch variables are integers are `i, j, k, m, n` and for reals `x, y, z` .

4.1.3 The prefix *n* should be used for variables representing the number of objects

```
nFiles, nSegments
```

This notation is taken from mathematics where it is an established convention for indicating the number of objects. A MATLAB-specific addition is the use of `m` for rows and `n` for columns in `mRows, nCols`.

4.1.4 As a convention, plurals should use a full suffix instead of 's'

All variable names (except for those represent the number of objects) should be singular. There should not be two variables which only differ by a final letter 's'. The suggested alternative is to used suffixes such as List, Array, Set, etc.

```
pointArray, fileNameList
```

4.1.5 Variables representing a single entity number can be suffixed by *No* or prefixed by *i*

The 'No' notation is taken from mathematics where it is an established convention for indicating an entity number

```
tableNo employeeNo
```

The 'i' prefix effectively makes the variables into iterators

```
iTable, iEmployee
```

4.1.6 Iterator variables should be prefixed with *i j k* etc.

Again, the notation is taken from mathematics where it is an established convention for indicating iterators.

```
for iFile = 1:nFiles
    :
end
```

It is preferred that iterators be longer than an single letter. This is especially true when using complex numbers, as i and j need to be reserved for the imaginary number. For nested loops, the iterator variables should be in alphabetical order and be helpful names.

```
for iFile = 1:nFiles
    for jPosition = 1:nPositions
        :
    end
end
```

4.1.7 Negated boolean variables should be avoided

A problem arises when such a name is used in conjunction with the logical negation operator ˜ as this results in a double negative. It is not immediately apparent what `~NotFound` means. Instead, use `isFound`. Do not use `isNotFound`.

4.1.8 Acronyms, even if normally uppercase, should be mixed or lowercase

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML, etc. which obviously is not very readable. When the name is connected to another, the readability is seriously reduced; the word following the abbreviation does not stand out as it should
Use `html, isIfoSpecific, checkGpsSector`.
Avoid `hTML, isIFOSpecific, checkGPSSector`.

4.2 Constants

4.2.1 Named constants should be uppercase using the underscore to separate words

This is common practice in the C++ development community. Although MathWorks may appear to use lowercase names for constants (i.e. 'pi'), such built-in constants are actually functions.

```
MAX_ITERATIONS, TRUE, FALSE
```

4.2.2 Constants can be prefixed by a common type name

This fives additional information on which constants belong together and what concept the constants represent

```
IFO_H1, IFO_H2, IFO_L1
```

4.3 Structures and Classes

4.3.1 Structure and class names should begin with a capital letter

This usage is consistent with C++ practice, and it helps to distinguish between structures and ordinary variables

```
NewChan   StartGps
```

4.3.2 Field and method names should not include the structure or class name

This repetition of the object name is not needed, and just adds to confusion.
Use `Segment.Length` and avoid `Segment.segmentLength`.

4.3.3 Where possible, define a struct in one command

Use the `struct` function. Otherwise, fields will be added in the order they are encountered which can be different for different paths through the code

4.4 Functions

4.4.1 Names of functions should be written in lowercase

It is clearest to have the function and its m-file names be the same. Using lowercase avoids potential filename problems in mixed operating system environments

```
getname(), computetotalwidth()
```

Do not use underscores between words. This makes the function names overly long and starts a confusion with constant naming.

4.4.2 Functions should have non-cryptic meaningful names

There is an unfortunate MATLAB tradition of using short and often somewhat cryptic function names. This is likely due to old 8-character limits in DOS. This concern is no longer relevant and the tradition should usually be avoided in user functions to improve readability
Use `computetotalwidth()` and avoid `compwid()` An exception is the use of abbreviations and acronyms widely used in mathematics

```
 max(), gcd(), ttgauge()
```

4.4.3 Functions with a single output may be named for the output

This is common practice in MATLAB code from MathWorks.

```
mean(), standarderror()
```

4.4.4 The prefixes *get/set* should generally be reserved for accessing an object or property

This is a general practice of MathWorks and a common practice in C++ and Java development.

```
getobj(), setobj()
```

4.4.5 The prefixes *calc* and *compute* can be used for methods where something is computed

Consistent use of these enhances readability. They give the reader the immediate clue that this is a potentially complex or time-consuming operation

```
calcweightedaverage(), computespread()
```

4.4.6 The prefix *find* can be used in methods where something is looked up

Give the reader the immediate clue that this is a simple lookup method with a minimum of computations involved. Consistent use of the term enhances readability and is a good substitute for get

```
findfirstevent(), findblargestwindow()
```

4.4.7 The prefixes *init* or *initialize* can be used where an object or a concept is established

The full prefix (please spell with a z) may be too long for many applications. In these cases, the shorter ’init’ can be used.

```
initfiltermatrix(), initializetimearray()
```

4.4.8 The prefix *is* should be used for boolean functions

This is a common practice in MathWorks MATLAB code, as well as C++ and Java.

```
isdone(), isfull()
```

There are a few alternatives to the ’is’ prefix that fit better in some situations. These include ’has’, ’can’ and ’should’ prefixes

```
hasdata(), canevaluate(), shouldsort()
```

4.4.9 Complement names and prefixes should be used for complement operations

One can reduce the complexity by using the symmetry of name pairs

```
get/set, create/destroy, start/stop, insert/delete, inc/dec
old/new, beg/end, up/down, min/max, next/prev, read/write
```

4.4.10 Avoid unintentional shadowing (i.e. make names unique between packages)

In general function names should be unique within project. If this is difficult, consider preceding every file/function in a package with a package-specific 2-3 letter prefix. Shadowing (having to or more functions with the same name) increases the possibility of unexpected behavior or error. Names can be checked for shadowing using `which -all` or `exist`.

4.5 General

4.5.1 Names of dimensioned variables and constants should have a units suffix

Adding units suffixes helps avoid the almost inevitable mixups. This should certainly be done in cases of easy confusion (angles, strain). Common abbreviations can be used for the suffixes.

```
incidentAngleRad,  incidentForceNm.
```

4.5.2 Abbreviations in names should be avoided

Using whole words reduces ambiguity and helps to make the code self-documenting
Use `computearrivaltime`, not `comparr()`
Domain-specific phrases that are more naturally known through their abbreviations or acronyms should be kept abbreviated. Even these cases might benefit form a defining comment near their first appearance

```
html, cpu, c
```

4.5.3 Consider making names pronounceable

Names that are at least somewhat pronounceable are easier to read and remember

4.5.4 All names should be written in English

The MATLAB distribution is written in English, and English is the preferred language for international development

5 Coding Style

5.1 Variables and Constants

5.1.1 Variables should not be reused unless required by memory limitations

Modern compilers are quite good at optimizing memory usage. Reusing scratch variables may even be counter-productive and lead to unexpected results then the reused variable is not redefined properly.

5.1.2 Each variable should be declared in a separate statement

While readability may be enhanced by grouping variables of the same type in a common statement, it defeats goals of simplicity and maintainability.

5.2 Globals

5.2.1 The use of global variables should be minimized

Instead, variables should be passed by arguments in function calls. Structures can be used to group variables together. Use the `persisitent` statement to maintain variables at the same value within a function between calls to the function.

5.2.2 Use of global constants should be minimized

Consider using a function (as MATLAB does) for global constants. This controls where they are defined.

5.3 Loops

5.3.1 Loop-indexes arrays should be initialized before the loop

This improves loop speed and helps prevent bogus values if the loop does not execute at all.

```
result = zeros(1,nEntries);
for index = 1:nEntries
    result(index) = foo (index)
end
```

5.3.2 The use of break and continue in loops should be minimized

These constructs can be compared to `goto` and they should only be used if they prove to have higher readability than their structured counterpart.

5.4 Conditionals

5.4.1 Complex conditional expressions should be avoided

Instead, use temporary logical variables. By assigning logical variables to expressions, the program gets automatic documentation. The construction will be easier to read and to debug. Replace

```
if (value>=lowerLimit)&(value<=upperLimit)&~ismember(value,...
valueArray)
    :
end
```

with the following

```
isValid = (value >= LowerLimit) & (value <= upperLimit);
isNew   = ~ismember(value, valueArray);
if (isValid & is New)
    :
end
```

5.4.2 The conditional expressions *if 0 if 1* shall be avoided

With revision control systems, there is no need to keep disabled code in files in case it is needed in the future. Also, do not leave debug code in code files. It eventually becomes very unclear if it is needed or not, and it is never kept up-to-date.

5.4.3 The constant should be listed first when checking equivalence

MATLAB typically prevents the typo `if(myVar = CONSTANT)`. However, it is safest to protect against this by listing the constant first `if(CONSTANT == myVar)`.

5.4.4 A switch statement should include the otherwise condition

Leaving the `otherwise` out is a common error, which can lead to unexpected results. The following format should be used

```
switch (condition)
    case ABC
        statements;
    case DEF
        statements;
    otherwise
        statements;
end
```

5.4.5 The *switch* variable should be a string

Character strings work well in this context and they are usually more meaningful than enumerated classes.

5.5 General

5.5.1 Parentheses should be used to define the order of evaluation

MATLAB has documented rules for operator precedence, but who wants to remember the details? If there might be any doubt, use parentheses to clarify expressions. They are particularly helpful for extended logical expressions.

5.5.2 Explicit numbers should be used sparingly in expressions

Explicit numbers are 4555, 4.33, 1E-19. Numbers that are subject to change should be implemented as named constants instead. If a number does not have an obvious meaning by itself, readability is enhanced by introducing a named constant instead. It can be much easier to change the definition of a constant than to find and change all of the relevant occurrences of a literal number in a file.

5.5.3 Floating point constants should be written with a digit before the decimal place

This adheres to mathematical conventions for syntax. Also, 0.5 is more readable than .5; it is not likely to be read as the integer 5.
Use `THRESHOLD = 0.5`; Avoid `THRESHOLD = .5`.

5.5.4 Floating point comparisons should allow for imprecision

Binary representations can cause trouble, as seen in this example

```
shortSide = 3;
longSide = 5;
otherSide = 4;
longSide^2 == (shortSide^2 + otherSide^2)
ans =
   1
scaleFactor = 0.01;
longPart = (scaleFactor*longSide)^2;
shortPart = ((scaleFactor*shortSide)^2 + (scaleFactor*otherSide)^2);
longPart == shortPart
ans =
   0
```

The reason is found by taking the difference

```
longPart - shortPart
ans =
   4.3368e-19
```

When comparing floating-point numbers for equivalence, always allow a small difference due to round-off and truncation. One can use the built-in 'eps' constant for this. An example is below

```
abs(longPart - shortPart) < eps
```

6 Formatting

6.1 Layout

The purpose of layout is to help the reader understand the code. Indentation is particularly helpful for revealing structure.

6.1.1 Content should be kept within the first 80 columns

80 columns is a common dimension for editors, terminal emulators, printers and debuggers. Files that are shared between several people should keep within these constraints. Readability improves if unintentional line breaks are avoided when passing a file between programmers. By default, the MATLAB Editor limits lines to 75 columns.

6.1.2 Lines should be split at graceful points

Split lines occur when a statement exceeds the suggested 80 column limit In general

- Break after a comma or space
- Break after an operator
- Align the new line with the beginning of the expression on the previous line

For example

```
totalSum = a + b + c + ...
           d + e;
function (param1, param2, ...
          param3)
setText (['Long line split' ...
          'into two parts.']);
```

6.1.3 Basic indentation and tabs should be 4 spaces

Good indentation is probably the single best way to reveal program structure. Indentation of 4 is the default used by the current MATLAB Editor, and thus is the easiest to standardize on. Indentation of 2 is sometimes suggested to reduce the number of line breaks required to stay within 80 columns for nested statements, but MATLAB is usually not deeply nested.

6.1.4 Indentation should be consistent with the MATLAB Editor

The MATLAB Editor provides indentation that clarifies code structure and is consistent with recommended practices for C++ and Java.

6.1.5 Tabs used for indents in editors should insert spaces

The MATLAB Editor option 'Tab key inserts spaces' inserts 4 spaces when the Tab key is pressed. This ensures that code files appear correctly in other editors, which may have the Tab set to another value. Note that some files (Makefiles) required true Tabs, instead of spaces.

6.1.6 A line of code should contain only one executable statement

This improves readability and allows Just-In-Time acceleration.

6.1.7 All if, for and while statements should be on multiple lines

It is possible to write short if, for and while statements on a single line. However, this has the disadvantages that there is no indentation format cue and it can not be easily expanded. Thus, it should be avoided.

6.2 White Space

White space enhances readability by making the individual components of statements stand out.

6.2.1 Assignment and binary logic operators should be surrounded by spaces

This includes `= & | > <`. Using space around the assignment character provides a strong visual cue separating the left and right hand sides of a statement. Using space around the binary logical operators can clarify complicated expressions

```
simpleSum = firstTerm+secondTerm;
complexLogical = (oneVal==testVal) & (numVal<limitVal);
```

6.2.2 Commas in a list should be followed by a space

These spaces can enhance readability

```
foo(alpha, beat, gamma)
```

6.2.3 Keywords should be followed by a space

This practice helps to distinguish keywords from functions.

6.2.4 Blocks of statements should be separated by one blank line

Enhance readability by introducing white space between blocks of statements.

6.2.5 Alignment may be used to enhance readability

Code alignment can make split expressions easier to read and understand. This layout can also help to reveal errors.

```
weightedPopulation = (doctorWeight*nDoctors) + ...
                     (lawyerWeight*nLawyers) + ...
                     (chiefWeight*nChiefs);
```

7 Comment Style

In MATLAB all comment lines start with % . Comments include the help text in the header of an M-file, later descriptive comments and Program Design Language (PDL)

7.1 Function Header

MATLAB makes special use of the header of a source code file, or M-file. This includes H1 text (used by `lookfor`) and help text (used by `help`).

7.1.1 Function headers must have comments for help text

The help text is the first group of consecutive comment lines following the function definition line. It is displayed when the `help` command is used.

7.1.2 Function headers should always have a descriptive H1 line

The first comment line immediately following the function definition line is the H1 line. This is the first line in the help text and is also displayed in directory lists in MATLAB. As such, it should include the name of the function and be as descriptive as possible. As this text is used in searches, include any likely search words.

7.1.3 The H1 line should start with the function name in uppercase

This follows MathWorks practice, which is intended to make the function name prominent. Of course, it is in opposition to their standard on function names, which is all lowercase.

7.1.4 Help text must show the function parameters

To be most useful, the help text should echo the function definition line. It should also describe each input and output parameter, including optional ones. In particular, it should discuss any special requirements for the input parameters

7.1.5 The last line of help text should restate the function definition

This allows the user to glance at the help printout to see the input and output arguments.

7.1.6 The help text should not be cluttered with non-descriptive text

It is common to include copyright lines and change history in comments near the beginning of a function file. To accommodate the MATLAB help text, there should be a blank line between the help text comments and the other header comments to avoid them being displayed with the `help`' command.

7.1.7 The function header shall have any keywords for the revision control system

The function header (after the help text) should provide a comment line with the %Id% keyword. This will be expanded by most revision control systems and will document where the file is stored.

7.2 Program Design Language

Program Design Language, or PDL, can be used to document the process design and logic of a function. It is essentially a structured form of commenting. It should be used before the actual coding, and be updated and corrected as the function is coded and tested. The book ”Code Complete” by Steve McDonnell has a good description of what is intended. This standard can be used for non-MATLAB code that is created (C, Perl) as well.

7.2.1 PDL should use English-like statements that precisely describe specific operations

Here is an example of PDL from McDonnell’s book.

```
% SET the default status
% GET the message based on the error code
% IF the error code is valid
%   DETERMINE the processing method
%   IF doing iterative processing
%       PRINT the error message interactively
%    DECLARE success
%   ELSE doing batch processing
%       IF the batch message file opens properly
%        LOG the error message to the batch file
%    CLOSE the file
%    DECLARE success
%    ENDIF
%   ENDIF
% ELSE the message is not valid
%   NOTIFY the user that an internal error has occurred
% ENDIF
```

7.2.2 PDL should avoid syntactic elements from the target language

The PDL should be at a higher level of abstraction/design than the code. It should in fact be independent of the actual programming language to be used. This include ignoring whether an instruction is coded locally or is performed through a function call.

7.2.3 PDL should describe intent, not implementation

Describe the meaning of the approach in PDL rather than how the approach will be implemented in the target language.

7.2.4 PDL should be written in enough detail that its relation to the code is clear

The PDL should be in enough detail that generating code from it should be straight-forward. If the PDL is at too high a level, it will gloss over problematic details in the design.

7.2.5 PDL text should be indented to match the indenting of the logic

This means 4-space tabs for all ifs, loops, etc. Note the this can not strictly be followed for the highest level, as MATLAB code starts in column 1, while comment text can only start in column 3 to allow a space after the comment character %.

7.2.6 All comments shall have the % character in column 1

By placing the comment character (%) in the first column, it is clear visually where the comments are.

7.2.7 PDL should be identified by putting the first word of each line in uppercase

This technique will serve to clarify which comments are PDL and which are other explanatory notes. It can also lead to the use of consistent sets of such words to add precision to the design description. Use IF/ELSE/ENDIF, LOOP/ENDLOOP, WHILE/ENDWHILE, SET/CLEAR. You can use the same prefixes used in function names to start lines (FIND, GET, CALCULATE, INITIALIZE).

7.2.8 PDL and its associated code should be grouped into screen-sized chunks

Experience has shown that instead of interleaving PDL and source code, instead keep the PDL together in blocks. Size the PDL blocks such that each PDL block (and the code which implements it listed afterward) take up about one editor screen's worth of space. This makes it easy to compare the PDL and the source code.

7.3 Other Comment Types

7.3.1 Non-PDL comments which explain the design may be included

Often, there is a need for additional comments that do not fit into declarative PDL statements. It is good to have comment blocks which

- describe the high-level processing of a function
- detail the equations used
- explain the rationale for a particular algorithm

7.3.2 Important variables should be documented near the start of the file

It is standard practice in other languages to document variables where they are declared. Since MATLAB does not use variable declarations, this information can be provided in comments.

7.3.3 Important constants should be documented near the start of the file

This is also standard practice in other languages.

8 MATLAB-specific Guidelines

The MATLAB language is similar to C in syntax. Its design avoids some of the pitfalls of C, but introduces some additional complexities that must be dealt with.

8.1 Functions

8.1.1 Functions shall test for the correct number of inputs and outputs

Because MATLAB does not have function prototypes, a function can make no assumptions about how it is called. Thus, at minimum, it should reject calls which do not have valid numbers of input and output parameters. The MATLAB functions `nargchk` and `nargoutchk` can be used for these checks. When such an error is detected, the function should echo back the function definition to the user and do no further processing.

8.1.2 Functions should test all inputs for type and range

MATLAB functions can also not assume that inputs are either of the correct type nor within the expected range of values. Each function should verify that the inputs are valid before using them.

8.1.3 Functions should use *varargin* and *varargout* for optional inputs and outputs

For optional inputs and outputs, MATLAB functions should use the `varargin` and `varargout` arrays to pass these parameters. Functions may wish to use these for all inputs and outputs.

8.1.4 Functions should support non-scalar inputs when possible

One of the great powers of MATLAB is that most functions can handle the same inputs as scalars, vectors or matrices. When possible, user functions should offer the same flexibility. This also means that all functions need to be sensitive to the dimensionality of inputs, and reject calls will pass inputs of incorrect dimensionality.

8.2 Arrays

8.2.1 Single-dimension vectors should be defined with one row, many columns

By default, a variable defined as a 1-dimensional vector in MATLAB has one row, multiple columns. This should be followed when initializing vectors, for example:

```
vectData = zeros(1,5)
for ii=1:5
    vectData(ii) = ii;
end
```

9 Legacy Code

Every project will have code which has been implemented outside the scope of the current standards regarding software design and coding.

9.1 Legacy Packages

9.1.1 A test suite must be in place before making significant changes to legacy packages

One of the great dangers with modifying existing code is breaking one part while fixing another. Before starting on substantial modifications, a baseline test suite must be created and results stored. This will help ensure that the modified code does not introduce new errors. The modified code must be run against the test suite to ensure unmodified requirements are still met.

9.1.2 When substantially modifying a legacy package, all functions should meet the naming standards

By naming all functions according to the standard, the legacy package will be more consistent with other packages already at the standard.

9.2 Legacy Functions

9.2.1 Bring legacy file up to the standard if more that one-third is to be modified

This creates a mechanism by which (over time) many of the legacy files will be brought up to the standard, but without imposing too great a workload at any one time. If a substantial amount of a function is to be changed, the entire design should be done in PDL to ensure that it is understood.

9.2.2 All blocks of code added to legacy functions must have PDL comments

This is another method whereby a legacy function (over time) is brought up to the standard.

References

[1] *MATLAB Programming Style Guidelines*, Richard Johnson, Datatool, October, 2002

[2] *LIGO Data Analysis System - Numerical Algorithms Library Specification and Style Guide*, LIGO-T990030-07-E, February, 2000 (draft)

[3] *Code Complete*, Steve McConnell, Microsoft Press, 1994

[4] *Using MATLAB*, Version 6, The MathWorks, Inc., 2002