CHAPTER 2

■ ■ ■

MATLAB Style

This chapter provides guidelines and recipes for suggested style elements to make your code and tools more understandable and easier to use.

When structuring a function, we have specific guidelines. The comments should be clear and descriptive and follow the formatting guidelines set by your institution. The same goes for naming conventions. In addition, we recommend supplying “built-in” inputs and outputs, that is, example parameters, so the function can be completely executed without any input from the user. These additional demo forms should be listed with the different syntaxes you create for the function.

Documenting your code goes beyond adding a header and some comments to your code. MATLAB now allows you to integrate HTML help into your toolboxes that displays in the browser along with MATLAB’s documentation. You can also use the publishing utility to create comprehensive technical reports. Incorporating these features into your style guidelines from the beginning will save you a lot of work when you want to release your toolbox to others.

2.1 Developing Your Own MATLAB Style Guidelines

Problem

Each engineer in your group has their own favorite naming and whitespace styles. Whitespace is the blank spaces and lines that are left between code elements. Some writers like the code very compact; others add a lot of whitespace to set things off, which may make the code longer. When people work on each other’s code, you end up with a mishmash that makes the code more difficult to read.

Solution

Develop and publish your own style guidelines. MATLAB can help enforce some guidelines, such as tab sizes, in the preferences.

How It Works

We recommend the classic book *The Elements of MATLAB Style* by Richard K. Johnson as a starting point for developing your own style guidelines. Many of the recommendations are generic to good coding practice across programming languages, and others are specific to MATLAB, such as using publishing markup syntax in your comments. The book addresses formatting, naming, documentation, and programming.

We deviate from the book's recommendations in a few ways. For one, we prefer to capitalize the names of functions. This distinguishes your custom functions from built-in MATLAB functions in scripts. In another, we like to use single-letter variables for structures, rather than long camel case names such as MyFancyDataStructure. However, the key to clear MATLAB code, which is also emphasized in Johnson's text, is to treat MATLAB code like compiled code. Be mindful of variable types, use parentheses even when MATLAB doesn't explicitly require them, and write plentiful comments. This means, generally, at least one comment with each code block and loop explaining its purpose. If you've left a blank line between code bits or indented a section of code, there should be a comment.

For instance, when a variable value is a double, indicate this with a decimal point. This avoids confusion that the parameter may be an integer.

Replace

```
value = 1;
```

with

```
value = 1.0;
```

In `if` statements, always use parentheses. If you ever want to port the code to another language in the future, this saves you time, and it makes the code clearer and easier to read for programmers versed in multiple languages.

Replace

```
if thisIsTrue && thatIsTrue
```

with

```
if (thisIsTrue && thatIsTrue)
```

You should always avoid "magic numbers" in your code, which are easy to use when quickly typing out a function to test a concept. This is a number value that is typed in, such as to a logical statement, instead of assigned to a variable. Take the time to define a properly named variable and add a comment with the source of the number.

Replace

```
if (value > 2.0 || value < 0.0)
```

with

```
if (value > minValue || value < maxValue)
```

With the advent of color-coding IDEs, such as MATLAB's editor, adding a lot of whitespace to delineate code sections has fallen out of favor in style guidelines. Generally, one line of blank space is enough between blocks of code. We suggest adding an additional line of whitespace between the end of a function and the start of a subfunction. You shouldn't put whitespace between lines of code that are closely related.

Some programmers prefer to align blocks of code on their equal signs. This can be helpful, especially when coding sets of equations from a reference. However, it can also be tedious to maintain when code is under active development. If you like this style, you may prefer to wait on adding the aligning space until the function has passed internal code review and is ready for release. In our code, we generally align on equals signs only within smaller blocks as delineated by comments or whitespace.

Consider:

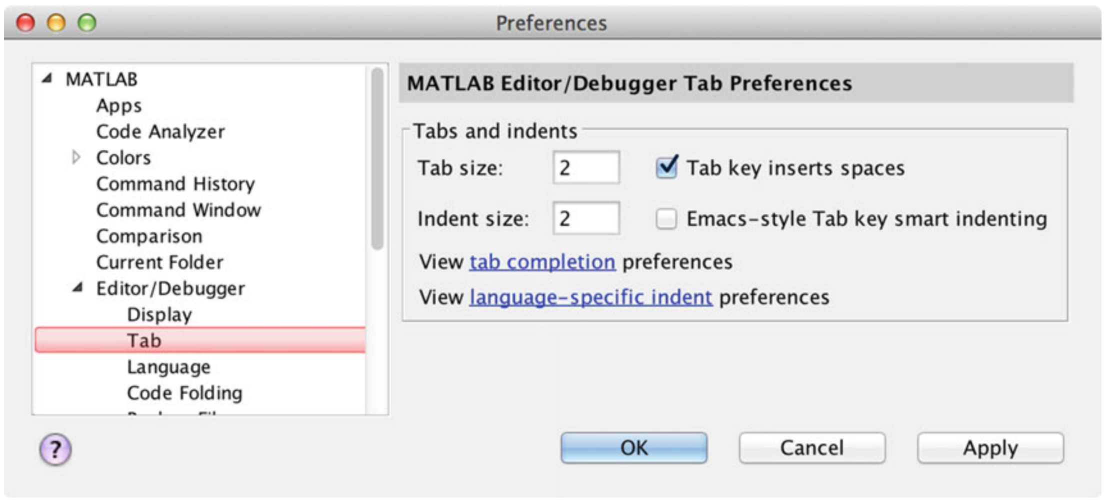
```
% Initialization
myVar1 = linspace(0,1);
b = 1.0;

% Calculation
[result1, result2] = MyFunction(myVar1,b);
plotH = plot(myVar1,result2);
```

This could be aligned in multiple ways, such as

```
% Initialization
myVar1 = linspace(0,1);
b      = 1.0;

% Calculation
[result1, result2] = MyFunction(myVar1,b);
plotH              = plot(myVar1,result2);
```

or, if aligning across the blocks, as

```
% Initialization
myVar1             = linspace(0,1);
b                  = 1.0;

% Calculation
[result1, result2] = MyFunction(myVar1,b);
plotH              = plot(myVar1,result2);
```



MATLAB Editor/Debugger Tab Preferences

Tabs and indents

Tab size: 2

✓ Tab key inserts spaces

Indent size: 2

☐ Emacs-style Tab key smart indenting

View tab completion preferences

View language-specific indent preferences

Figure 2.1: *Tab preferences with size of two and spaces option checked.*

In our code for this book, you will see the former, per-block style of alignment.

Another consideration with whitespace is tab sizes. Some guidelines recommend larger tabs of four or eight spaces, arguing that MATLAB code is rarely deeply nested. We routinely write a lot of deeply nested code so our internal guideline is for two spaces. When you set the tab size in the MATLAB preferences, and set it to insert spaces for tabs, you can use the Smart Indent feature to easily highlight and update code blocks. Figure 2.1 shows the tab preferences pane in MATLAB R2014b, on a Mac.

We prefer to use uppercase for function names (`MyFunction`) specifically to distinguish them from the lowercase function names of the built-in MATLAB functions. Otherwise, we use camel case (`myVariableName`) for variables and often single-letter or very short names for structures. For index variables, we tend to use `k` to avoid confusion with the variables `i` and `j` and their association with imaginary numbers. We follow the standard convention of capitalizing constant names, that is, for the Earth's gravitational constant, `MU_EARTH`.

In other words, you need to establish naming conventions for the following:

- Function names
- Variable names
- Structure names
- Index variables
- Constants

Additional naming conventions might include standard prefixes or suffixes for certain types of files or variables. One example is using the letters "RHS" in the name of a function that provides dynamics for integration, that is, the right-hand side of the equations when the derivatives are on the left. The word "Demo" is helpful in the name of a script that demonstrates a particular function or feature. You should be consistent about the order of variable name elements. For example, if `r` means radius and the second element is the name of the planet, then use `R_EARTH` and `R_MOON`. Don't make the second `MOON_R`. The order should be consistent throughout your code base.

Further rules could address the names of boolean variables or the use of verbs in function names. The most important step is to create a set of conventions for your organization and write them up, or create some function templates, so that your engineers write consistent code.

The guidelines we have described here and use throughout this book are summarized as follows:

Naming – Naming guidelines

- **Function names** – Use camel case for function names with the first letter capitalized. The first word is ideally an action verb or "RHS."
- **Script names** – If the script is a demo of a particular function or set of functions, append "Demo" to the name.
- **Variable names** – Use camel case for variable names with a lowercase first letter.
- **Constants** – Use all uppercase with underscores to identify constants.
- **Variable name length** – Most variable names should be at least three characters. This helps enforce uniqueness and makes the variables more readily distinguishable. Exceptions include commonly used data structures, index variables, and when replicating equations from a text where single-letter variable names are standard and easily recognizable to someone in the field.
- **Index variables** – When using a single index variable, use `k`; when using two, `j` and `k`; for additional variables, use `l`, `m`.

Doubles – Always use a decimal point when typing out a double value.

Magic numbers – Avoid magic numbers in your code; prefer the use of a variable to specify a number.

Comments – Always add a comment describing the source or rationale for a hard-coded number in your code.

If statements – Always use parentheses around the conditional portion of IF statements.

Tabs – Use a tab size of two spaces and set MATLAB to insert spaces for tabs. Use Smart Indent to enforce consistent tabs before committing files.

Blank lines – Use one blank line between most code sections and two blank lines between subfunctions.

Alignment – Align code on the equals sign only within the code block, as separated by blank lines.

Guidelines for function headers are addressed in the next recipe.

2.2 Writing Good Function Help

Problem

You look at a function a couple months (or years) after you wrote it, or a colleague wrote it, and find it has only one cryptic comment at the top. You no longer remember how the function works, what it was supposed to do, or exactly what your comment means.

Solution

Establish a format for your function headers and stick to it. Use the publishing markup to enable you to generate good-looking documentation from the m-file.

How It Works

Write the header for your function at the top, using the publishing markup. This means that the very first line should start with a section break, %%, and include the name of your function, as that will be the title of the published page. This line should also include a one-sentence summary of the function; this must be in the first non-empty comment line of the file and is also termed the "H1" line. This summary can be used automatically by MATLAB when generating `Contents.m` files for your folders and by the `lookfor` function, which searches files on the path for keywords.

Document inputs and outputs separately using section titles. Indicate the type or size of the variable and provide a description. Using two spaces between the comment sign % and line's text will generate monospaced text, which we use for the input and output lists. We have developed the following keys to indicate the variable type and size:

- {} – Cell array
- `(1,1)` – Scalar value
- `(:)` or `''` – String
- `(:,:)` – Matrix of variable size
- `(1,:)` – Row of variable length
- `(:,1)` – Column of variable length
- `(m,n)` – Matrix with row and column sizes (m,n) that must match other inputs or outputs
- `(.)` – Data structure

- `(:)` – Data structure array
- `(*)` – Function handle

Always include a copyright or authorship notice. Take credit for authoring your code! The standard is to start with the initial year the function is created, then add a range of years when you update the function, that is, Copyright (c) 2012, 2014–2015. The "c" in parentheses approximates the actual copyright symbol. After copyright, the next line should state "All Rights Reserved." Add a contract number, project number, or distribution statement if pertinent. Add a blank line between the main header and the copyright notice to suppress it from the command-line help display.

We show an example in the following for a function which computes a dot product column-wise for two matrices. Note that this will still be legible in the command window output of `help Dot`, with the first % of the cell breaks suppressed. We use the * markup for a bulleted list. The output will always be one row which is indicated in the size key.

FUNCTION HEADER EXAMPLE

```
%% DOT Dot product of two arrays.
%% Forms
%  d = Dot( w, y )
%  d = Dot( w )
%% Description
% Dot product with support for arrays. The number of columns of w and y can be:
%
% * Both > 1 and equal
% * One can have one column and the other any number of columns
%
% If there is only one input the dot product will be taken with itself.
%% Inputs
%  w  (:,:)  Array of vectors
%  y  (:,:)  Array of vectors
%% Outputs
%  d  (1,:)  Dot product of w and y
%% See also
% Cross
```

When published to HTML, this will appear as follows, ignoring the generated Contents section:

DOT Dot product of two arrays.

Forms

```
d = Dot( w, y )
d = Dot( w )
```

Description

Dot product with support for arrays. The number of columns of w and y can be:

- Both > 1 and equal
- One can have one column and the other any number of columns

If there is only one input the dot product will be taken with itself.

Inputs

```
w  (:,:)  vector
y  (:,:)  vector
```

Outputs

```
d  (1,:)  Dot product of w and y
```

See also

Cross

Finally, remember to describe any plots created or files generated, that is, "side effects." It's also a good idea to identify if a function uses persistent or global variables, which may require a `clear` command to reset. The following list summarizes the parts of the header, in order:

1. **H1 line** – Start with a single line description of the function.
2. **Syntax** – List the syntax supporter.
3. **Description** – Provide a more detailed description. Describe any built-in demos, default values for parameters, persistent or global variables users need to be aware of, and any "side effects" including plots or files saved. Indicate if a function will request input from the user.
4. **Inputs** – List the inputs with a size/format key. Include units, if applicable.
5. **Outputs** – List the outputs as for inputs.
6. **See also** – List any related functions.
7. **Reference** – If applicable, list any references.
8. **Copyright** – Include a copyright notice. There should be a blank line between the rest of the header and the copyright notice.

2.3 Overloading Functions and Utilizing `varargin`

Problem

You want to reuse a section of code you have written, but you may have to use it in different situations or extract additional data for it in some circumstances but not others.

Solution

You can overload functions in MATLAB easily and implicitly. This means calling the same function with different inputs and outputs. `varargin` and `varargout` make it simple to manage variable-length input and output lists.

How It Works

MATLAB allows you to overload a function in any way you would like, inside the file that defines it. This applies to the inputs and the outputs. There is generally a trade-off between writing the clearest code you can, with a single calling syntax, and avoiding duplication of code. Perhaps there are intermediate variables which may be useful as outputs in some cases, or you want to provide backward compatibility with an older syntax. When creating libraries for numerical computations, there always seem to be additional syntaxes that are useful. We recommend the following when overloading functions:

- Use `varargin` and `varargout` when possible and rename the variables with descriptive names as close to the top of the function as you can.
- Be sure to document all input and output variants clearly in the header. Adding another optional input or output and neglecting to document it is the #1 reason for out-of-date headers.
- Use comments to clearly identify what the outputs are when you are renaming them to match the function's syntax, or use `varargout`.
- Clear the function outputs if you are creating a plot and they are not needed, to avoid unnecessary printing to the command line.

The following example highlights the use of these guidelines. We often use functions with a string "action" defining multiple input variations by name. This provides additional clarity beyond depending on the input number or type to select an overloaded method.

FUNCTION OVERLOADING

```matlab
% d = OverloadedFunction('default data');
% OverloadedFunction('demo');
% [y,d] = OverloadedFunction('update',x,d);

function varargout = OverloadedFunction( action, varargin )

```

```
switch action
  case 'default data'
    d = DefaultData;
    varargout{1} = d;

  case 'demo'
    d = DefaultData;
    x = linspace(0,1);
    y = OverloadedFunction('update',x,d);
    figure('name','OverloadedFunction Demo');
    plot(x,y);

  case 'update'
    x = varargin{1};
    d = varargin{2};
    y = Update(x,d);
    varargout{1} = y;
    varargout{2} = d;

end
```

2.4 Adding Built-in Inputs and Outputs to Functions

Problem

You would like to provide default values for some optional inputs or provide a short demonstration of how a function works.

Solution

Add built-in inputs and outputs to your function using an enumerated input or with `nargin`. This can include a full demo that calls the function and generates plots, as appropriate.

How It Works

Built-in inputs provide an example set of parameters that produce output. In many cases, we provide an input range that can create a plot demonstrating the computation performed in the function. In the case of MATLAB, you must explicitly handle input options in the code, as you can't add a default value in the function definition itself.

One convention that we find useful is to allow for an empty matrix, [], to be entered for an input to use its default value. This allows you to request a default for one input but provide values for subsequent inputs. The following example shows both a demo that creates a plot and a default value for a constant.

```
function output = MyFunction( variable, constant )

if (nargin == 0)
  % perform demo
  variable = linspace(0,100);
  output = MyFunction( variable );
  return;
end
if (nargin < 2 || isempty(constant))
  % default value of constant
  constant = 2.05;
end
```

Notice that the built-in demo, which is performed when there are no inputs at all, calls the function itself and then returns. This makes the demo also a built-in test. The code to generate the built-in outputs, which could be a text report to the command line or a plot, generally comes at the end of the function. This enables you to create the built-in outputs with inputs the user specifies and not just the built-in inputs. For instance, there might be alternative values of the constant. Note that in the following output generation example, the name of the figure is specified including the name of the function, which is exceedingly helpful if you routinely generate dozens of plots during your work.

```
... body of function with calculations ...

if (nargout==0)
  % Default output is a plot
  figure('Name',sprintf('Demo of %s',mfilename))
  plot(variable, output)
  clear output
end
```

■ **TIP** Assign a name to figures that you create. Include the name of the function or demo for clarity. The name will be displayed in the title bar of the figure and in MATLAB's Windows menu.

Writing all your functions this way has several advantages. For one, you are showing valid ranges of the variables up front, without requiring a reader to refer to a separate test function or demo in another folder. Having this hard data available every time you open the function helps keep your code and your comments consistent. Also, you have a test of the function, which you can easily rerun at any time right from the editor. You can publish the function with execution turned on which will perform the demo and include the command-line output and plots right in the HTML file (or LaTeX or Word, if you so choose.) All of this helps reduce bugs and documents your function for other readers or yourself in the future.

Following this guideline, here is the general format we follow for all functions in this book:

1. Detailed header
2. Copyright
3. Function definition
4. Default inputs
5. Function demo that calls itself
6. Code body with calculations
7. Default output

Note that no final `return` statement is necessary.

In summary, we have enabled the following usages of this function by adding default inputs and outputs:

```
output = MyFunction( variable, constant );
output = MyFunction( variable );    % uses default value of constant
MyFunction;                         % performs built-in demo
MyFunction(variable, constant);     % creates a plot for the given input
```

2.5 Adding Argument Checking to Functions

Problem

You would like to check arguments to make sure the user has entered valid values. Perhaps the function only works for a certain range of values or only positive inputs. You would like to give the user specific feedback on why a function has failed due to a known input issue rather than let the function throw an error.

Solution

Add argument validation. The MATLAB document has details about the syntax.

How It Works

We create the following function, `ArgCheckFun`, and use the new `arguments` feature available since R2019b. This allows you to explicitly define the type of each function argument. You can define default values for arguments and supply validation functions. A table of predefined functions exists, including `mustBePositive` and `mustBeNegative`.

```matlab
function [x,c] = ArgCheckFun(a,b,c)

arguments
  a (2,1) double
  b (2,1) double
  c (1,:) char
end

x = a'*b;
```

Now let's run a few tests.

```
>> ArgCheckFun
Error using ArgCheckFun
Invalid input argument list. Not enough input arguments.
Function requires 3 input(s).
```

This first test fails as expected.

```
>> ArgCheckFun(1,1,1)

ans =
     2
```

The second does not fail, even though the arguments are not what is specified in the argument block. Note that the third input is unused except for being passed back as an output, so the wrong form does not cause an error.

```
>> ArgCheckFun(rand(3,1),1,1)
Error using ArgCheckFun
Invalid input argument at position 1. Value must be vector with 2
    elements.
```

The third does fail, as it should, as the first input has too many elements.

```
>> [x,c] = ArgCheckFun(rand(2,1),rand(2,1),'test1')

x =
    0.2652

c =
    'test1'
```

The last text works.

There may be cases where this is useful. However, if you properly document your arguments in the header, you won't have much need for this feature.

2.6 Adding Dot Indexing

Problem

You would like to use dot indexing with a function that outputs a data structure.

Solution

Create a function with a single data structure output. Dot indexing is a new feature available by default.

How It Works

Create the following function:

```
function d = StructFun(v)
d = struct('x',v,'y',0);
```

Now run two tests, passing 2 for the value v, and in the first test, accessing x, and in the second, accessing y.

```
>> StructFun(2).x

ans =
     2

>> StructFun(2).y

ans =
     0
```

You can see that you can now access a field directly. It is not necessary to use a data structure variable as an intermediate step if you only need a single field's value.

2.7 Smart Structuring of Scripts

Problem

You write a few lines of code in a script to test some idea. Can you figure out what it does a year later?

Solution

Treat your scripts like functions, and structure them well. Take the time to follow a template.

How It Works

A script is any collection of commands that you put in a file and execute together. In our toolboxes, we treat scripts as demos of our toolbox functions, and therefore as instructional. Here are some guidelines we recommend when creating scripts:

Create help – Help headers are not just for functions; write them for your scripts too. Will you remember what this script does in a year? Will someone else in your company be able to understand it? Write a detailed description including a list of any files required or generated.

Use publishing markup – Create cells in your scripts (using %%) to delineate sections. Write detailed comments after the section headings. Publish your script to HTML and see how it looks. You can even add equations using LaTeX markup or insert images.

Initialize your variables – Take care to fully initialize your variables or you could have conflicts when you run multiple scripts in a row. This especially applies to data structures and cell arrays. See the recipes for data types in Chapter 1 for the correct way to initialize different variables.

Specify a directory for saved files – Make sure you are saving any data into a particular location and not just wherever the current directory happens to be.

Our scripts follow the following pattern. Cell breaks are used between the sections.

1. Detailed header using publishing markup.
2. Copyright notice.
3. User parameters, meant to be changed between runs, are grouped at the top.
4. Constants are defined.
5. Initialize plotting arrays before loops.
6. Perform calculations.
7. Create plots.
8. Store outputs in files, if desired.

A complete example, which can be executed, is shown as follows:

ScriptDemo.m

```matlab
%% This is a template for a script layout.
% A detailed description of the script includes any files loaded or generated
% and an idea of what data and plots will be created. We will calculate a sine
% or cosine with or without scaling of the input. The script creates one plot
% and saves the workspace to a file called Demo.mat.
%% See also
% sin, cos

%% User parameters
param1   = 0.5;
nPoints = 50;
useSine = false;

%% Constants
MY_CONSTANT = 0.25;

%% Calculation loop
yPlot = zeros(2,nPoints);
x     = linspace(0,4*pi,nPoints);
for k = 1:nPoints
  if (useSine)
    y = sin( [1.0;param1]*x(k) + MY_CONSTANT );
  else
    y = cos( [1.0;param1]*x(k) + MY_CONSTANT );
  end
  yPlot(:,k) = y;
end

%% Plotting
figure('Name','DEMO');
plot(x,yPlot);

%% Save workspace to a file
saveDir = fileparts(mfilename('fullpath'));
save(fullfile(saveDir,'Demo'))
```

We can verify that the data is stored by clearing the workspace and loading the mat-file after the demo has been run.

```
>> clear all
>> ScriptDemo
>> clear all
>> load Demo.mat
>> who

Your variables are:

MY_CONSTANT  nPoints      useSine      y
k            param1       x            yPlot
```

2.8 Implementing MATLAB Command-Line Help for Folders

Problem

You have a set of folders in your code base and you would like users to easily navigate them as they can the built-in MATLAB library.

Solution

Placing `Contents.m` files in each folder can provide metadata for the contents of the folders, and this can be displayed on the command line.

How It Works

Command-line help isn't just for functions and scripts. Folders can also have help in the form of a contents listing, which includes the function names and a single-line description of each. Toolboxes can also provide documentation in response to a `ver` command with a toolbox-level contents listing. This information is provided in a `Contents.m` file that consists entirely of comments.

The built-in *Contents Report* can generate `Contents.m` files for you. It can also check and fix existing `Contents.m` files. This will automatically use the "H1" line, or the first line of the header in the function or script. In Recipe 2.2, we provide an example of a function header that includes this line. This report is accessed, along with other reports, in the Current Folder window, using the pop-up menu. To read more and learn how to run the report on your operating system, see the MATLAB help topic "Create Help Summary Files." It's typically accessed from the context menu in the Current Folder window.

Version information isn't limited to a single Contents file per toolbox; it is generated by special lines inserted into the top of any `Contents.m` file:

```
% Version xxx dd-mmm-yyyy
```

You can also add a descriptive line above the version information and add subheadings to groups of files. For example, consider the output from the `codetools` directory included in MATLAB:

```
>> help codetools
  Commands for creating and debugging code
  MATLAB Version 8.4 (R2014b) 08-Sep-2014

  Editing and publishing
    edit                       - Edit or create a file
    grabcode                   - Copy MATLAB code from published HTML
    mlint                      - Check files for possible problems
    notebook                   - Open MATLAB Notebook in Microsoft Word (on
        Microsoft Windows platforms)
    publish                    - Publish file containing cells to output file
    snapnow                    - Force snapshot of image for published
        document

  Directory tools
    mlintrpt                   - Run mlint for file or folder, reporting
        results in browser
    visdiff                    - Compare two files (text, MAT, or binary) or
        folders
```

As with the header of a function, there can be no blank lines in the Contents file, only comments. This is shown in an example from Chapter 7 of this book, the double integrator, where we have added letters of the alphabet as section breaks.

```
% MATLAB/Ch06-DoubleIntegrator
%
% D
%    DoubleIntegratorSim - Double Integrator Demo
%
% P
%    PDControl           - PDCONTROL Design and implement a PD
   Controller in sampled time.
%    PlotSet             - PlotSet Create two-dimensional plots from a
   data set.
%
% R
%    RHSDoubleIntegrator - RHSDoubleIntegrator Right hand side of a
   double integrator.
%    RungeKutta          - RungeKutta Fourth order Runge-Kutta
   numerical integrator.
```

```
%
% T
%    TimeLabel            - TimeLabel Produce time labels and scaled
   time vectors
```

Figure 2.2 shows how to access the Contents Report for this folder from the command window. If a `Contents.m` file doesn't already exist, the report will create one for you, which will then be available from the command line via `help FolderName`.

The actual report is shown in Figure 2.3. You can see that there are links to edit the `Contents.m` file, such as for adding version information, fixing the spacing, or fixing all problems. The report will detect if you have changed the H1 description line of the function, and it conflicts with the text in the Contents file. It allows you to update those descriptions with a single click, avoiding any copy/paste.

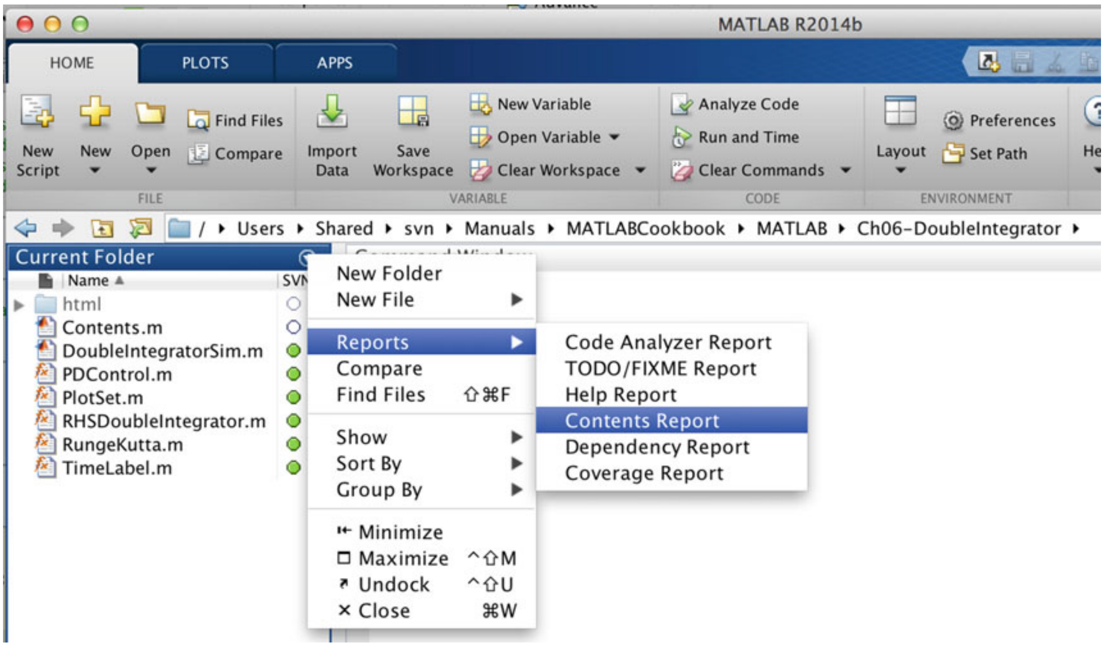
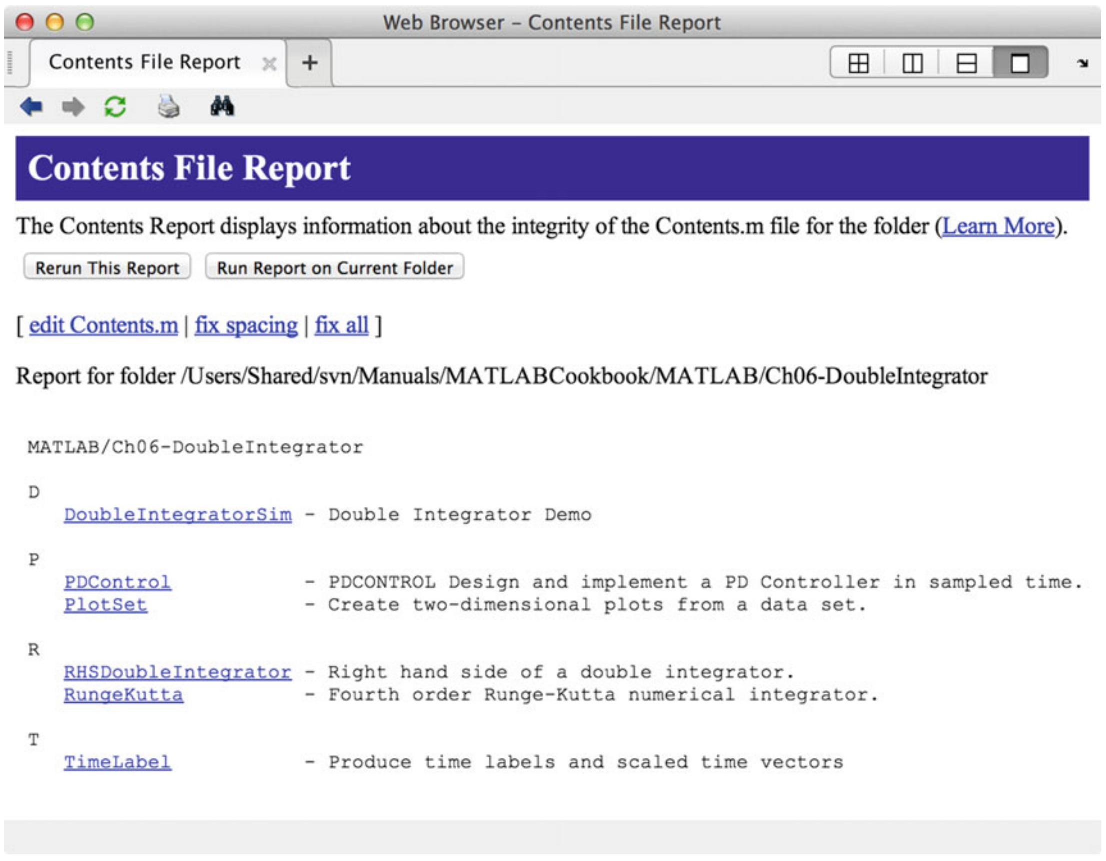


Figure 2.2: *Access the Contents Report on double integrator.*


Web Browser – Contents File Report
Contents File Report
Rerun This Report
Run Report on Current Folder

Contents File Report

The Contents Report displays information about the integrity of the Contents.m file for the folder (Learn More).

[edit Contents.m | fix spacing | fix all]

Report for folder /Users/Shared/svn/Manuals/MATLABCookbook/MATLAB/Ch06-DoubleIntegrator

```
MATLAB/Ch06-DoubleIntegrator

D
   DoubleIntegratorSim - Double Integrator Demo

P
   PDControl           - PDCONTROL Design and implement a PD Controller in sampled time.
   PlotSet             - Create two-dimensional plots from a data set.

R
   RHSDoubleIntegrator - Right hand side of a double integrator.
   RungeKutta          - Fourth order Runge-Kutta numerical integrator.

T
   TimeLabel           - Produce time labels and scaled time vectors
```

Figure 2.3: *Completed double integrator contents report.*

2.9 Publishing Code into Technical Reports

Problem

You are creating a report based on some analysis you are doing in MATLAB. You are laboriously copying and pasting code snippets and figures into your report document. You discover a bug in your code, and you have to do it all over again.

Solution

The publishing feature in MATLAB allows you to run a script and automatically generate a document containing the results, including images of each figure generated and the code itself, with text and equations that you insert. These reports can be easily regenerated when you change your code.

How It Works

The publishing features allow you to generate HTML, LaTeX, Word, and PowerPoint documents from your code. These documents can display the code itself as well as command-line output and plots. You can even capture snapshots of your figures during loops and include equations using LaTeX markup. Every programmer should become familiar with these features. We highlight the main features in the following.

The very first section at the top of your file gives a title to the published document. The comments which follow in your header will be published as discussed in Recipe 2.2. Having a good header is important since this can be displayed at the command line, up until the first blank line of your function. However, you can also add more sections, text, equations, and images throughout your code. MATLAB will automatically generate a table of contents of all the sections and will insert the generated plots and command-line output in each section.

You need to be careful about putting section breaks inside loops, since this will produce a snapshot of any figures at every iteration. This could be a desired behavior if you want to capture the evolution of a figure, but could also accidentally produce hundreds of unwanted images. The following is an example script, MemoExample.m, created to demonstrate publishing.

CREATE A TECHNICAL MEMO FROM YOUR CODE

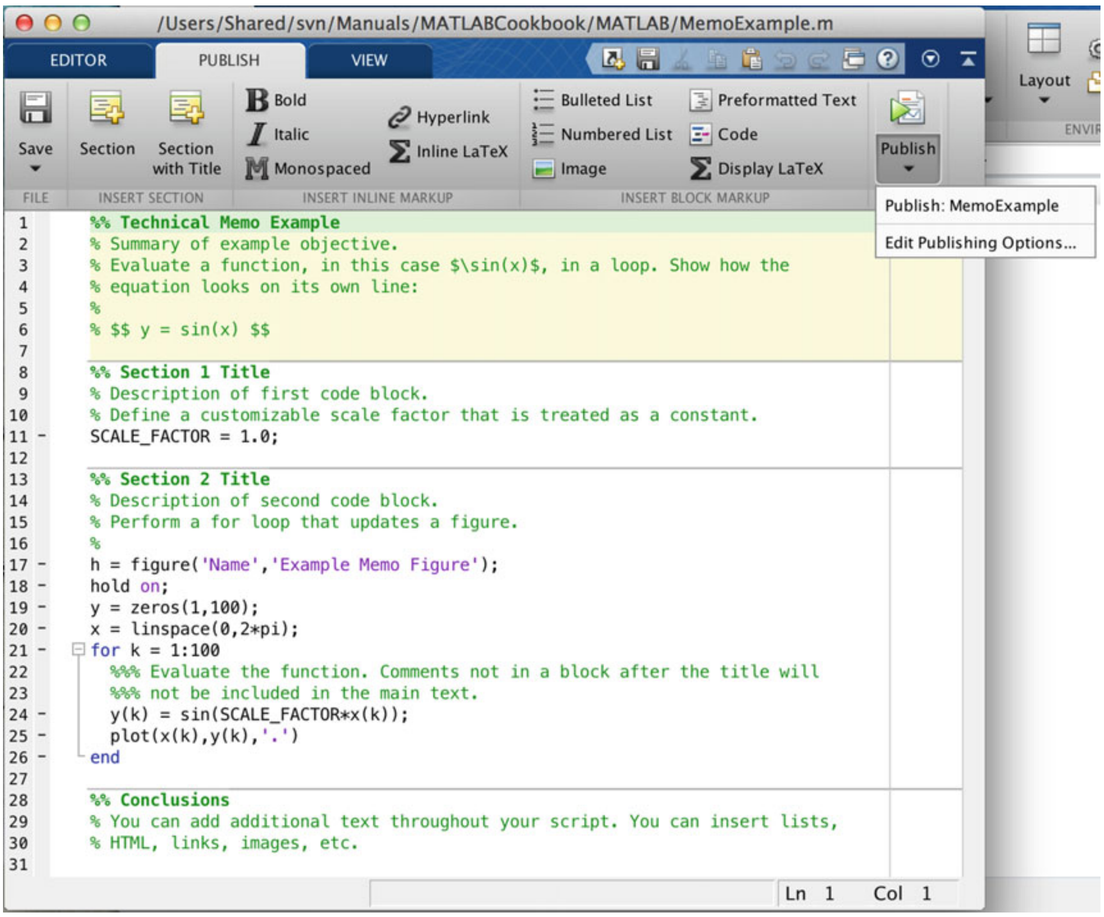
MemoExample.m

```
%% Technical Memo Example
% Summary of example objective.
% Evaluate a function, in this case $\sin(x)$, in a loop. Show how the
% equation looks on its own line:
%
% $$ y = sin(x) $$

%% Section 1 Title
% Description of first code block.
% Define a customizable scale factor that is treated as a constant.
SCALE_FACTOR = 1.0;

%% Section 2 Title
% Description of second code block.
% Perform a for loop that updates a figure.
%
h = figure('Name','Example Memo Figure');
hold on;
y = zeros(1,100);
x = linspace(0,2*pi);
for k = 1:100
        %%% Evaluate the function. Comments not in a block after the
            title will
        %%% not be included in the main text.
        y(k) = sin(SCALE_FACTOR*x(k));
        plot(x(k),y(k),'.')
end

```

```
%% Conclusions
% You can add additional text throughout your script. You can insert
   lists,
% HTML, links, images, etc.
```

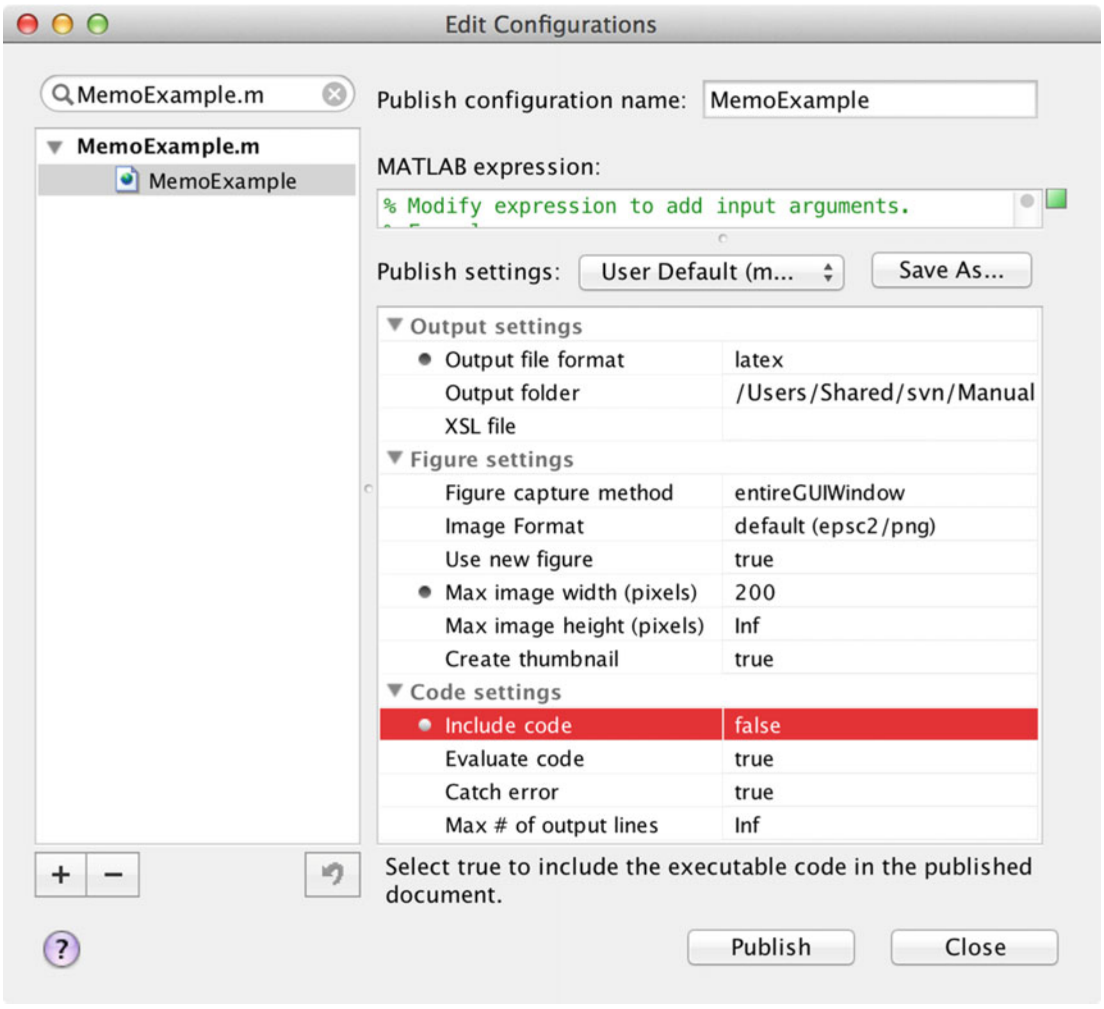
Figure 2.4 shows this script in the publishing tab of the MATLAB editor, with the pop-up menu opened to access the publishing options.

There are a number of settings that apply to publishing. You can save a set of settings with a name and easily reuse it for all of your files. The default settings for code are to both evaluate it and include the source code in the published document, but these may be turned off independently. To create a technical memo from a script without including the source code itself, you set the "include code" option to *false*. You can set maximum dimensions on figures and select the format – JPEG, PNG, bitmap, or TIFF. You can even specify a MATLAB



```
%% Technical Memo Example
% Summary of example objective.
% Evaluate a function, in this case $\sin(x)$, in a loop. Show how the
% equation looks on its own line:
%
% $$ y = sin(x) $$

%% Section 1 Title
% Description of first code block.
% Define a customizable scale factor that is treated as a constant.
SCALE_FACTOR = 1.0;

%% Section 2 Title
% Description of second code block.
% Perform a for loop that updates a figure.
%
h = figure('Name','Example Memo Figure');
hold on;
y = zeros(1,100);
x = linspace(0,2*pi);
for k = 1:100
  %%% Evaluate the function. Comments not in a block after the title will
  %%% not be included in the main text.
  y(k) = sin(SCALE_FACTOR*x(k));
  plot(x(k),y(k),'.')
end

%% Conclusions
% You can add additional text throughout your script. You can insert lists,
% HTML, links, images, etc.
```

Figure 2.4: *Preparing to publish a script in the editor.*



Publish configuration name: MemoExample

MATLAB expression:

```
% Modify expression to add input arguments.
```

Publish settings: User Default (m...

▼ Output settings	
● Output file format	latex
Output folder	/Users/Shared/svn/Manual
XSL file	
▼ Figure settings	
Figure capture method	entireGUIWindow
Image Format	default (epsc2/png)
Use new figure	true
● Max image width (pixels)	200
Max image height (pixels)	Inf
Create thumbnail	true
▼ Code settings	
● Include code	false
Evaluate code	true
Catch error	true
Max # of output lines	Inf

Select true to include the executable code in the published document.

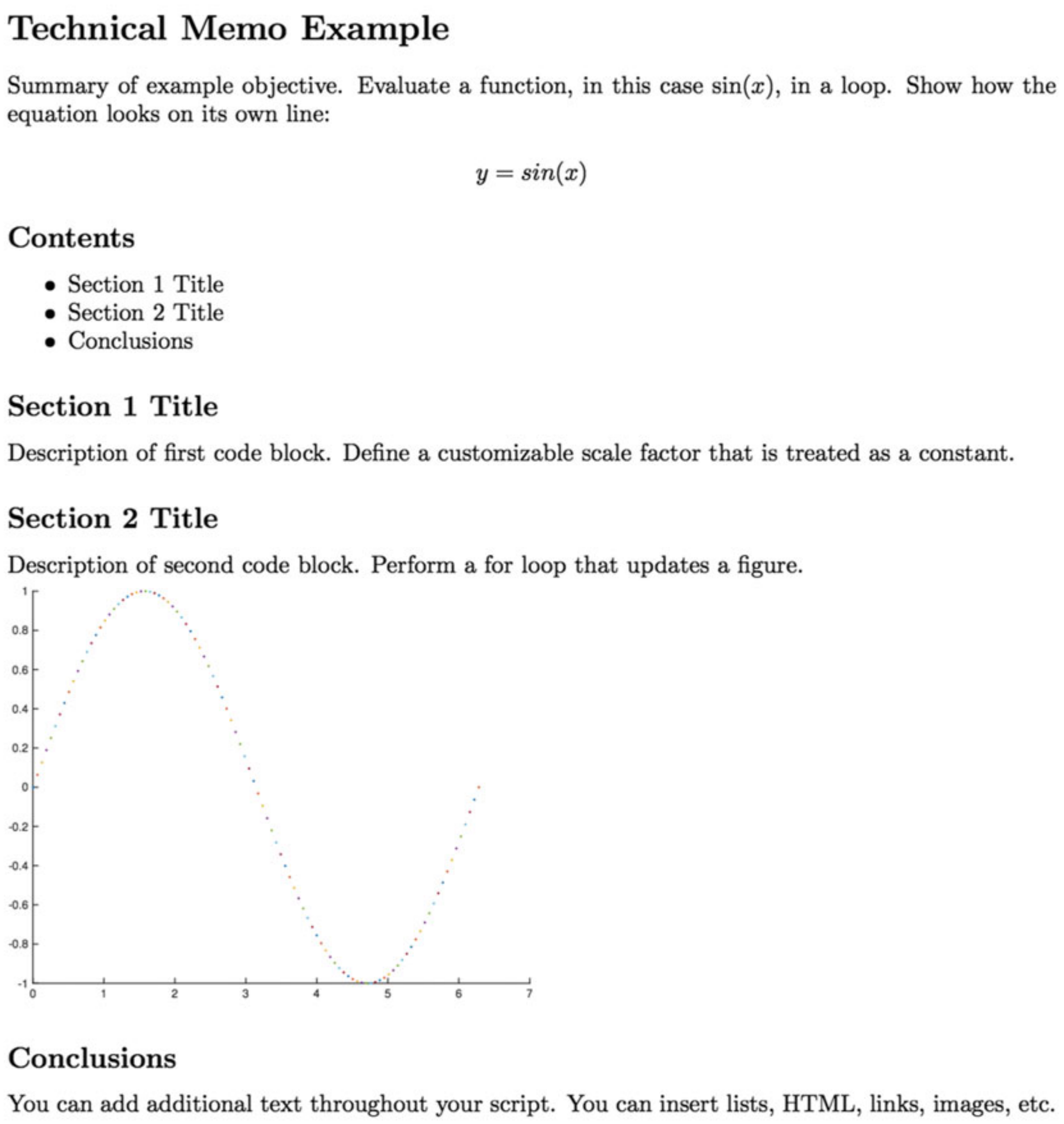
Figure 2.5: *Editing the publish settings for a file.*

expression for a function to include input arguments, rather than just running it as a built-in demo.

Figure 2.5 shows the settings window with PDF selected as the output type. Note the `Save As...` button which allows you to save settings. We set the maximum width of the figure to 200 pixels to enable the memo to fit on one page, for the purposes of this book.

Figure 2.6 shows a LaTeX memo generated and compiled for the preceding listing published without the code, with the figure generated in a loop. Note the table of contents, equation, and insertion of the graphic. We had to remove some extra `\vspace` commands that MATLAB added to the LaTeX to fit the memo on one page.

Figure 2.6: *Technical memo published to LaTeX and compiled to PDF.*



Technical Memo Example

Summary of example objective. Evaluate a function, in this case sin(x), in a loop. Show how the equation looks on its own line:

$$y = sin(x)$$

Contents

- Section 1 Title
- Section 2 Title
- Conclusions

Section 1 Title

Description of first code block. Define a customizable scale factor that is treated as a constant.

Section 2 Title

Description of second code block. Perform a for loop that updates a figure.

Conclusions

You can add additional text throughout your script. You can insert lists, HTML, links, images, etc.

2.10 Integrating Toolbox Documentation into the MATLAB Help System

Problem

You would like to write a user's guide and provide it with your toolbox.

Solution

If you write HTML help files, you can in fact include them with your toolbox when you distribute it, and the help will show up in MATLAB's help browser.

How It Works

You are not limited to the command-line help when providing documentation for your code or toolbox. MATLAB now provides an API for writing HTML documentation and displaying it to users in the help browser. You can write an entire HTML manual and include published versions of your demos.

In order to integrate your HTML help files into the MATLAB help system, you need to generate a few XML files. One provides a top-level table of contents for your toolboxes. Another provides a list of the demos or examples. The third identifies your product. The help topics to read are "Display Custom Documentation" and "Display Custom Examples." The help for third-party products is displayed in a separate section of the MATLAB help browser entitled "Supplemental Software." The files you need to generate are

info.xml – Identify your documentation

helptoc.xml – Table of contents

demos.xml – Table of examples

The MATLAB documentation describes the XML tags you need and provides template documents. Comments can be included within the files using standard HTML comments with <!-- and -->.

The main purpose of the info.xml file is to provide a name for your toolbox, identify it as a toolbox or blockset, and provide a path to the remaining HTML documentation. The following is an example for our Recipes code.

EXAMPLE INFO.XML

```
<productinfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="optional">
    <?xml-stylesheet type="text/xsl"href="optional"?>

    <matlabrelease>R2019b</matlabrelease>
    <name>MATLAB Recipes</name>
    <type>toolbox</type>
    <icon></icon>
    <help_location>Documentation</help_location>
    <help_contents_icon>$toolbox/matlab/icons/bookicon.gif</
        help_contents_icon>

</productinfo>
```

The table of contents file, helptoc.xml, must provide a listing of all the HTML files in your help. This is accomplished with a `<tocitem>` tag that can be nested. You are generally providing a starting or main page for your toolbox, a getting started page, users guide pages, release notes, and further pages listing the functions provided. `<tocitem>` can have references to HTML anchors; they do not all need to refer to separate HTML files.

A small set of icons is included that can be displayed in the help contents. Consider the following helptoc.xml:

EXAMPLE HELPTOC.XML

```
<?xml version='1.0' encoding="utf-8"?>
<toc version="2.0">
<!-- First tocitem specifies top level page in Help browser Contents
    -->
    <tocitem target="index.html">Recipes Toolbox
        <!-- A Getting Started page is generally first -->
        <tocitem target="getting_started.html" image="HelpIcon.
            GETTING_STARTED">
            Getting Started
            <tocitem target="requirements.html">System Requirements</
                tocitem>
            <tocitem target="features.html">Features
                <!-- TOC levels may include anchor IDs -->
                <tocitem target="features.html#10187">Feature 1</
                    tocitem>
                <tocitem target="features.html#10193">Feature 2</
                    tocitem>
            </tocitem>
        </tocitem>
        <!-- There is a special icon for the User Guide -->
        <tocitem target="guide_intro.html"
            image="HelpIcon.USER_GUIDE">Recipes User Guide
            <tocitem target="setup.html">Setting Up</tocitem>
            <tocitem target="data_processing.html">Processing Data</
                tocitem>
            <tocitem target="verification.html">Verifying Outputs
                <tocitem target="test_failures.html">Handling Test
                    Failures</tocitem>
            </tocitem>
        </tocitem>
        <!-- The function reference is next with the FUNCTION icon -->
        <!-- First item is page describing function categories, if any -->
        <tocitem target="function_categories.html"
                 image="HelpIcon.FUNCTION">Function Reference
            <tocitem target="function_categories.html#1">Double Integrator
                <!-- Inside category, list the functions -->
                <tocitem target="function_1.html">function_1</tocitem>
                <tocitem target="function_2.html">function_2</tocitem>
                <!-- ... -->
            </tocitem>
            <tocitem target="function_categories.html#2">Aircraft
                <tocitem target="function_3.html">function_3</tocitem>
                <tocitem target="function_4.html">function_4</tocitem>
             </tocitem>
            <tocitem target="function_categories.html#3">Spacecraft
                <!-- ... -->
            </tocitem>
        </tocitem>
        <!-- Web links with the webicon.gif -->
        <tocitem target="http://www.psatellite.com"
                 image="$toolbox/matlab/icons/webicon.gif">
        Web Site (psatellite.com)
        </tocitem>
    </tocitem>
</toc>
```

This produces the contents listing in the help browser shown in Figure 2.7. The major icons helping to delineate the help sections are used. Anchor IDs are used for both features.html and function_categories.html. There is even a reference to an external website. Note that this means you need to have written the following HTML files:

- index.html
- getting_started.html
- requirements.html
- features.html
- guide_intro.html
- setup.html
- data_processing.html
- verification.html
- test_failures.html

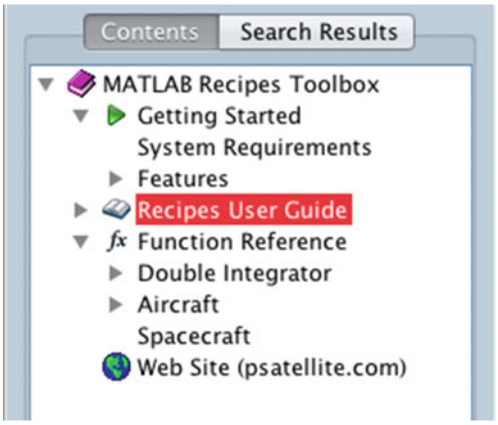


Figure 2.7: *Custom toolbox table of contents.*

- function_categories.html
- function_1.html
- function_2.html
- ...

Clearly, generating a function list for a large toolbox by hand could be cumbersome. At PSS, we have functions to generate this XML automatically from a directory, using `dir`. You can use the functional form of `publish` to publish your functions and scripts to HTML automatically as well.

The demos file is similar to the `toc` file in that it provides a nested list of demos or examples. There are two main tags, `<demosection>` and `<demoitem>`. Items can be m-files or videos. Published demos will display a thumbnail for one of the figures from the demo, if any exist; the thumbnail image will have the same name as the HTML file but a different extension. The demos are completely independent from the HTML table of contents, and you can implement an examples listing without creating any other HTML help pages.

Here is a short example from our Cubesat Toolbox that includes a published demo called `MagneticControlDemo`.

EXAMPLE DEMOS.XML

```
<?xml version="1.0" encoding="utf-8"?>
<demos>
   <name>CubeSat</name>
   <type>toolbox</type>
   <icon>$toolbox/matlab/icons/demoicon.gif</icon>
   <description>Contains all the demo files for the CubeSat</
       description>
   <website>
      <a href="http://www.psatellite.com">For more info see psatellite.
          com</a>
   </website>
```

```
    <demosection>
       <label>AttitudeControl</label>
       <demoitem>
          <label>MagneticControlDemo: Magnetic control demand analysis</
              label>
          <callback>MagneticControlDemo</callback>
          <file>../CubeSat/Demos/AttitudeControl/html/
              MagneticControlDemo.html</file>
       </demoitem>
    </demosection>
</demos>
```

Once you have created a set of HTML files, you can create a database that will allow MATLAB to search them efficiently. To do this, you use `builddocsearchdb` with a path to the folder containing your help files, that is, the same path you enter in your info.xml file. This function will create a subfolder called helpsearch containing the database. With this subfolder added to your help installation, users will get results from your documentation when they search in the help browser. Figure 2.8 shows a complete `Documentation` folder including the `helpsearch` database.

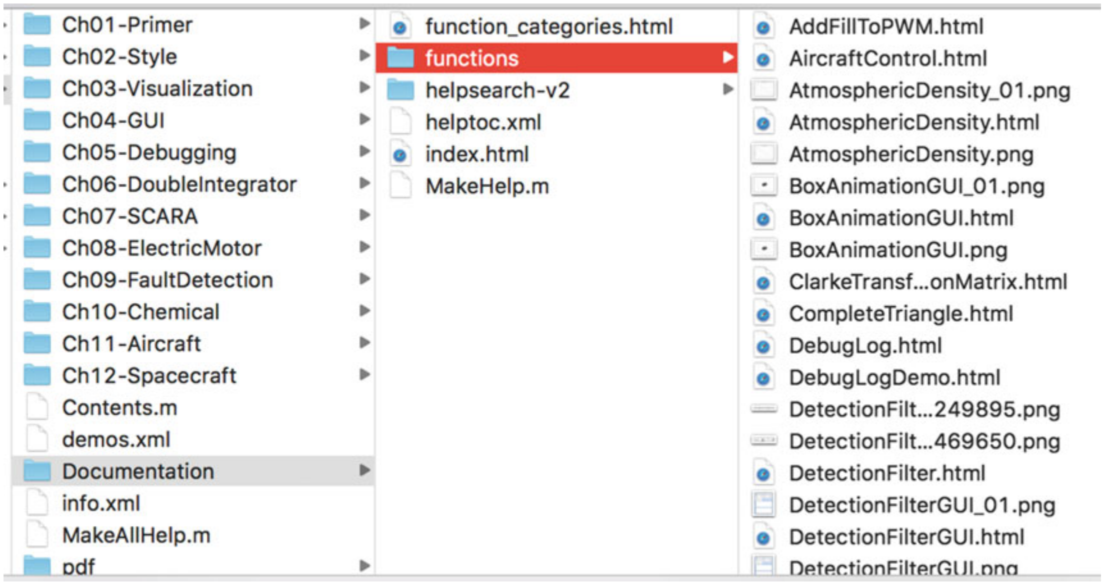


Figure 2.8: *Documentation including the search database for this toolbox.*

2.11 Structuring a Toolbox

Problem

You have a jumble of functions and scripts that you would like to organize into a toolbox that you can distribute to others.

Solution

A previous recipe showed how you can create or generate `Contents.m` files for individual folders in your toolbox. You can also create a top-level `Contents.m` file. We will describe our usual toolbox structure including the placement of these files.

How It Works

We have a fixed structure for our commercial toolboxes that is used by our build tools and testing routines.

- Group related functions together in folders.
- Place scripts in separate folders.
- Place script folders together in a Demos folder.
- Use the same name for the function folder and corresponding demos folder, for example, Mechanics/ being a folder with functions and Demos/Mechanics/ holding the corresponding demos.
- Organize folder groups into Modules or Toolboxes.

Once you create the help files as described in the previous recipes, they will appear in the directory structure as shown in the following – not in literal alphabetical order. Note that the published demos are stored in the html directories within the demo folders. We do not display them all, but every folder should have its own `Contents.m` file.

```
Module
|  Contents.m
|  Folder1
|    |    Contents.m
|    |    Function1.m
|  Folder2
|    |    Function2.m
|  Demos
|    |   Folder1
|    |     |   Function1Demo.m
|    |     |   html
|    |   Folder2
|    |     |   Function2Demo.m
|    |     |   html
|    |   CombinedDemos
|       |       |    SuperDemo.m
|       |       |    html
|    Documentation
|       |    demos.xml
|       |    info.xml
|       |    ToolboxHelp
|       |       |    helptoc.xml
|       |       |    GettingStarted.html
|       |       |    ...
```

You will note that there is a top-level `Contents.m` file within the Module, as the same level as the folders. MATLAB does not have any automated utility to make this for you. You can create one with a version line, the name of your toolbox, and any other information you would like to be displayed when the user types “help Module”; we generate a list of folders within the module using `dir`. Here is an example, noting that all lines in a `Contents.m` file are comments:

A Sample Contents.m

```
%  PSS Toolbox Folder NewModule
%  Version 2015.1      05-Mar-2015
%
%  Directories:
%  Folder1
%  Folder1
%  Demos
%  Demos/Folder1
%  Demos/Folder1
```

Your toolbox module will now appear when the user types `ver` at the command, for example:

```
>> ver
-----------------------------------------------------------------
MATLAB Version: 8.4.0.150421 (R2014b)
MATLAB License Number: 6xxxxx
Operating System: Mac OS X  Version: 10.9.5 Build: 13F1066
Java Version: Java 1.7.0_55-b13 with Oracle Corporation Java HotSpot(TM)
    64-Bit Server VM mixed mode
-----------------------------------------------------------------
MATLAB                                                Version 8.4
    (R2014b)
PSS Toolbox Folder NewModule                     Version 2015.1
```

Summary

In this chapter, we reviewed style guidelines for writing MATLAB code and highlighted some differences between styles for MATLAB and other languages. When establishing guidelines for your own toolboxes, consider the features you may want to use, such as automatic generation of contents files, publishing your results to HTML or Word, and even incorporating HTML help in the web browser. Also take the time to create proper headers and initialization when you generate code to avoid unpleasant surprises down the road!. Table 2.1 lists the code developed in the chapter.

Table 2.1: *Chapter Code Listing*

File	Description
`Dot`	Dot product header example
`MemoExample`	Example of a technical memo for publishing
`OverloadedFunction`	An internally overloaded function
`ScriptDemo`	Demo template for a script layout