SANDIA REPORT
SAND2007-2735
Unlimited Release
Printed May 2007

MatSeis Developer's Guide

Version 1.0

Lane C. McConnell and Christopher J. Young

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-2735
Unlimited Release
Printed May 2007

MatSeis Developer's Guide Version 1.0

Lane C. McConnell and Christopher J. Young
Next Generation Monitoring Systems
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0401

Abstract

This guide is intended to enable researchers working with seismic data, but lacking backgrounds in computer science and programming, to develop seismic algorithms using the MATLAB-based MatSeis software. Specifically, it presents a series of step-by-step instructions to write four specific functions of increasing complexity, while simultaneously explaining the notation, syntax, and general program design of the functions being written. The ultimate goal is that that the user can use this guide as a jumping off point from which he or she can write new functions that are compatible with and expand the capabilities of the current MatSeis software that has been developed as part of the Ground-based Nuclear Explosion Monitoring Research and Engineering (GNEMRE) program at Sandia National Laboratories.

ACKNOWLEDGMENTS

MatSeis was originally developed by Mark Harris, but the current version has contributions from many people, most notably Darren Hart and John Merchant. We are grateful to all of those that have worked on MatSeis, and to the many users who have given us valuable feedback that has helped to improve the package.

CONTENTS

INTRODUCTION

The MATLAB-based MatSeis software (Harris and Young, 1997) was developed as part of the Ground-based Nuclear Explosion Monitoring Research and Engineering (GNEMRE) program at SNL as a tool to aide in the investigation and prototyping of seismic algorithms. It was designed to take advantage of the broad array of basic utilities available in MATLAB as well as the advanced signal process utilities provided by the MATLAB Signal Processing Toolbox. MatSeis provides quick and easy access to data in an Oracle database or in properly formatted flatfiles, as well as an easy to understand graphical user interface. The current incarnation of MatSeis delivers many of the basic utilities routinely needed by seismologists for their research, but it is by no means complete, and it is not our intention to make the package comprehensive. That's where this guide comes in.

One of the most useful aspects of MatSeis and its associated tools is that it has readily expandable functionality. That is to say, once the user understands the basic data types and functionality associated with MatSeis, new functions are very easy to write. The purpose of this guide, then, is to explain how to write these new functions so that MatSeis users can customize the package to meet their needs. All that is required is a rudimentary knowledge of the MATLAB programming language and a bit of familiarity with using MatSeis.

With this in mind, our hope is that anyone working on seismic algorithm development and application prototypes will be able, with the help of this guide, to quickly learn the basic steps required to transform the theoretical concept into workable code. Essentially, the guide, much like the MatSeis program itself, should serve as a jumping off point to do this. It will, we hope, impart its readers with the programming skills and understanding to help make MatSeis a better tool.

FUNCTION 1: CUT OUT A SEGMENT OF A WAVEFORM

To demonstrate the ease of writing new functionality for MatSeis, let's write a function that will allow us to cut out a segment of a waveform and save that segment separately. And let us, for lack of creativity, call this function "cutseg" because it "cut"s a "seg"ment of a waveform out.

Begin with the normal MATLAB function header… the word "function" to indicate that what follows is a function, and the function's name (in this case "cutseg") followed by empty parenthesis to indicate that the function takes no input parameters. Note that there is also no assignment in the function header, a fact that indicates this is a void function with no return.

```
function cutseg()
```

There. Now we have a void function that takes no parameters. It doesn't do anything yet, but it's a start.

In order to cut out a segment from a waveform, two criteria need to be fulfilled. First, we must have one or more waveforms to work with, and second, we must have a time segment specified to cut out. Both of these would be done interactively using the MatSeis display. Let's make sure that we meet the first of these criteria (one or more waveforms have been selected) before going any further. As a check, we should add something like the following lines.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find selected waveforms.
%
N = waveform('index','select');
if isempty(N)
  errordlg('No waveforms selected!');
  return;
end
```

Firstly, any line beginning with the '%' character will be considered a comment by MATLAB and will therefore not be executed as part of the program. These lines are there only for the code developer and anybody who might have to read the code as an explanation of the purpose of an individual piece of the function. In fact, the '%' need not appear at the beginning of the line, but anything that follows it and is on the same line will be treated as a comment.

The piece of code above makes use of a couple of functions, one included as part of MATLAB and the other written by Sandia as part of the MatSeis package. To see whether a function is from MATLAB or MatSeis, type 'which function_name' at the MATLAB prompt, where function_name is the name of the function in question. The location of the file containing the function will be shown. Generally it does not matter whether a function to be used is from MATLAB, MatSeis, or a programmer's own suite--that is one of the reasons that MATLAB/MatSeis is such a powerful prototyping environment--but it can be useful to check if problems are encountered.

Returning to our example, the 'waveform' function is a MatSeis function (i.e. written by Sandia) that provides access to one of our five main data-types: waveforms, origins, arrivals, travel times, and measurements. There are also corresponding functions for the other 4 data types ('origin', 'arrival', 'traveltime', and 'measurement'). Each of these has many uses which are dictated by the specified parameters. The particular use of 'waveform' above finds the indices (hence 'index') of the selected waveforms (hence 'select') and returns them in an array which is assigned to the variable N. A full description of the waveform function can be seen by typing 'help waveform' at the MATLAB prompt, and you can use similar commands to get information about the other 4 data-type functions. We strongly encourage would-be MatSeis programmers to become familiar with these functions as they are the key to using our package to develop seismic applications.

But let's get back to our example. Once N has been returned, the code then checks whether that array has anything in it. To do this, it calls the MATLAB function `isempty()` on the array N. If there is nothing in the array, the message "No waveforms selected!" will be printed to the error dialog. Then the return that follows will cause a normal return to the invoking function.

Now comes another big decision. MatSeis will allow the user to either manipulate the original waveform or to create a new copy of the original, manipulate that, and display both. We use an 'if' statement to make this decision by checking how the parameter is set. MatSeis parameters are set in MatSeis config files, and you can check (or set) parameter values using the 'ms_par' MatSeis function.

If you do not specify a parameter in a MatSeis config file, this does not necessarily mean that the parameter has no value. Most of the parameters are set to default values in the function gui/ms_config.m.

Returning to our example, if the MatSeis parameter 'SIGPRO_REPLACE' is not set (the "~" indicates a "not") to true, then we have to make a new copy of the waveform to cut (i.e. the default behavior is to create a new waveform). Making a copy of a waveform (which includes the data, sample rate, station, channel, etc.) would involve several steps if we did it by hand, but we can do it in a single step using the 'copy' option of the 'waveform' function'. All we need to do is specify the index of the waveform that we wish to copy.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create new waveform?
%
if ~ms_par('SIGPRO_REPLACE')
  N = waveform('copy', N);
end
```

Note that we not only use N as a function parameter to specify the index of the waveform to copy but we also use the same variable on the left hand side to get the index of the new waveform. This isn't absolutely necessary. A different variable could have been used on the left hand side, but we chose to use N instead because we know that the original index will not be used after this point in our function and thus we can avoid introducing another variable if we recycle N.

Now that we've decided whether to edit a copy of the waveform or the original, it's time to actually edit it. To cut the waveform according to the limit lines set using the main MatSeis display, is an incredibly simple task, because the functionality to do so is already written. All we have to do is call the waveform function with the input parameters 'cut' to indicate that we want to perform the cut operation on the selected waveform, N to indicate which waveforms are selected (the new one!), and 'segment' to indicate that the portion of the waveform to be cut out is determined by the displayed limit lines. To do this, we use the code below:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Segment data.
%
waveform('cut', N, 'segment');
```

At this point, the new copy of the waveform has been cut, but we can't see that until we update the MatSeis display. To do this, we simply call the MatSeis function "ms_draw". This automatically redraws the waveforms and prints them to their correct locations in the MatSeis window.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot data.
%
ms_draw;
```

Now we can give ourselves a pat on the back. We've just written our very first function using a combination of MATLAB functions and MatSeis functions. Putting it all together, we should have code that looks something like the following:

```
function cutseg()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find selected waveforms.
%
N = waveform('index','select');
if isempty(N)
  errordlg('No waveforms selected!');
  return;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create new waveform?
%
if ~ms_par('SIGPRO_REPLACE')
  N = waveform('copy', N);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Segment data.
%
waveform ('cut', N, 'segment');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot data.
%
ms_draw;
```

FUNCTION 2: INTEGRATING A WAVEFORM

Okay. So it's probably seems impossible that *all* MatSeis Functions will be that easy to write. Well, there is some truth to that, but less than you might think given the rich set of functionality provided by MATLAB and MatSeis. Most functions can be broken down into enough smaller pieces for which functionality is already available that actually writing the code is fairly straightforward. In fact, the most difficult part of writing a function for most people tends to be translating a geophysical concept into a sequence of function calls, that is, to come up with a viable algorithm. Putting this algorithm into computer language is comparatively simple.

Let's look at a somewhat more intimidating function and show how, in reality, it isn't all that much more complicated.

As an example, let's write a function that we can use to integrate a waveform, e.g. to go from velocity to displacement. This may sound complicated, but it's really not that bad. We'll call the function "sp_integ".

```
function sp_integ()
```

Again, the function header consists of nothing more than the word "function" followed by the function's name and empty parenthesis. This is, therefore, going to be another void and parameterless function.

Also like before, the first step of this function is to locate the selected waveforms. We create an array called N which holds the indices of any selected waveforms. Then we check to make sure that N has something in it, because if it doesn't, then there are no waveforms selected (and we can't do much with nothing).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find selected waveforms.
%
N = waveform('index','select');
if isempty(N)
  errordlg ( 'No waveforms selected!' );
  return;
end
```

Now comes the new part. We have to actually process each of these waveforms and integrate them… separately. That means we're going to have to repeat the same steps on each of them. For this, we'll use a 'for' loop:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process each waveform.
%
for i = 1:size(N,1)
  n = N(i);
```

The function size(N,1) gets the number of rows in the matrix N, while size(N,2) would get the number of columns. In this case, N is an array (i.e. a matrix with only one column) that has the index of each selected waveform in a separate row, so size(N,1) is the number of selected waveforms. The line `for i = 1:size(N,1)` can be read as saying "for every instance of i from i equals one to i equals the number of selected waveforms". This colon notation specifies the first and last number that i should be as we travel through the loop. Unless otherwise specified, the increment is one. Therefore, this piece of code will dictate that every piece of code contained within it will be executed first with i equal to 1, then with i equal to 2, then with i equal to 3, all the way up to the point where i is the the number of the selected waveforms. In this way, we can iterate through each waveform that has been selected and integrate them one by one.

In the first line after the 'for' loop, we set the variable n to hold the index of the waveform we'll be integrating each time we go through the 'for' loop. In this manner, we only select the waveforms one at time.

Now we have to extract the data from the waveform at the nth index. To do this, we use the waveform function with the parameters of 'data' to indicate that's what we want to extract, 'n' to indicate which waveform we want to extract it from, and 'segment' to indicate that we only want the data falling in the specified time segment (which is specified on the MatSeis screen). Calling the waveform function with these parameters will return a vector holding all of the data, and we store this in a variable that we have chosen to call "data". The code to do so looks like this:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get data vector.
%
data = waveform ('data', n, 'segment');
```

Once we've tried to extract the data we're going to process, we have to make sure that we've actually extracted something. We check this with a quick 'if' statement. If the "data" variable we created isn't empty, then we continue on in the code.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If the data vector is not empty,
% continue to process the waveform.
%
if ~isempty(data)
```

So, assuming there's data to process, we can process it. This is the hard part… the part where the algorithm comes into play. What, we have to ask ourselves, is a simple way to integrate a waveform consisting of evenly spaced, linearly connected data points? Well, the simplest way to attain a decent approximation is to simply add up all of the data values then divide by the number of sampling points per unit of time.

To do this, we can use MATLAB's built in 'cumsum' function, which just gives a vector with the total sum (up to the given entry) of all previous entries in the vector which it is applied to. For instance, applying the 'cumsum' function to the vector [1 2 3 4 5] would return the vector [1 3 6 10 15]. This is exactly what we are looking for.

Now all that is left for us to do is to divide by the sampling rate (data points/time). We can get this for free using the waveform function again with the parameter 'samprate'. The code to integrate the waveform, then, appears as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Integrate data.
%
data = cumsum(data)/waveform('samprate', n);
```

Phew! That wasn't too bad, was it? And now we're back to familiar territory. We have to decide whether to replace the existing waveform with the one we've just processed or to print them both out to the screen. We do this just as before, storing the index of the copy of the waveform into the variable n (which otherwise continues to hold the index of the original waveform).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create new waveform?
%
if ~ms_par('SIGPRO_REPLACE')
  n = waveform('copy', n);
end
```

Note that at this point our new waveform is a copy of the unsegmented, selected waveform. That's not ultimately what we want, but it's quicker than creating a new waveform from scratch because we now only have to update the parts of the waveform data object that will be different for the final new waveform (e.g. the data, the start time, etc.).

Let's see how that can be done.

First, we want to update the data in our new waveform to be the integrated value stored in our data variable. The first line, waveform('data', n, data), essentially executes this. Notice the difference between the first 'data' and the second data. The first one indicates that we're going to construct a waveform from data, and the second data is the actual "data" variable holding the vector representing the processed information. The 'n' of course still indicates that all of this should be done to the waveform specified by the nth index.

The second line, waveform('time', n, 'segment'), updates the start time of the new waveform to be the start of the time segment.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Store data vector.
%
waveform('data', n, data);
waveform('time', n, 'segment');
```

Now we need to update the channel label for the new waveform to show that it is an integrated version of the original selected waveform. We set a variable called channel equal to a new string (i.e. a word or phrase) formed by appending the letters "INT" to the channel of the original selected waveform, then pass that variable to the waveform function.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Adjust channel name.
    %
    channel = sprintf('%sINT       ', ms_deblank(waveform('channel',n)));
    waveform('channel', n, channel);
```

The MATLAB function sprintf is used to write out a formatted string. The function ms_deblank is a MatSeis function that gets rid of any leading or trailing blanks in a string. We use this so that the new channel name will not have any unexpected blanks in it (i.e. we want things like "SHZINT" and not "SHZ INT"

Now we're pretty much done. The only thing left to do is make sure that we close any open loops or if statements that are still hanging around and print out the new, integrated waveform. We use the keyword 'end' to complete this. The first end closes the 'if' statement from way back where we decided to proceed only if there was data to process. The second one goes back even further to close the 'for' loop that allowed us to execute all of the code since on each selected waveform individually.

```
  end

end
```

Now we plot the data, just as we did with the first function we wrote, by calling the ms_draw function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot data.
%
ms_draw;
```

Gathering all of the code we've just written into one cogent piece, our function should look like this:

```
function sp_integ()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find selected waveforms.
%
N = waveform('index','select');
if isempty(N)
  errordlg('No waveforms selected!');
  return;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process each waveform.
```

```
%
for i = 1:size(N,1)
  n = N(i);

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Get data vector.
  %
  data = waveform('data', n, 'segment');

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If the data vector is not empty,
  % continue to process the waveform.
  %
  if ~isempty(data)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Filter data.
    %
    data = cumsum(data)/waveform('samprate',n);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Create new waveform?
    %
    if ~ms_par('SIGPRO_REPLACE')
      n = waveform('copy', n);
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Store data vector.
    %
    waveform('data', n, data);
    waveform('time', n, 'segment');

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Adjust channel name.
    %
    channel = sprintf ('%sINT       ', ms_deblank(waveform('channel',n)));
    waveform('channel', n, channel);

  end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot data.
%
ms_draw;
```

Notice that only about half of this code is actually new. The other half could be copied and pasted directly from the previous function. This is another secret of writing computer code; there are no rules to keep you from copying and pasting snippets of code that you've already written! In fact, it can save you a lot of time and energy.

FUNCTION 3: SHOW WAVEFORMS FOR AN ORIGIN

Now we're really rolling! Let's write another function, slightly more complicated than the previous couple. A common need in seismology when studying waveforms is to make sure that you're only looking at waveforms from a single origin, that is, waveforms who have arrivals associated with a given origin. We can write a function that will allow us to check this automatically. In fact, we can make it such that if any waveforms appear that don't have arrivals associated with a specified origin, then they are automatically deleted. Let's get started on this now.

Let's name our new function "showassoc_wf" to indicate that it will "show associated waveforms", and create the header like before.

```
function out = showassoc_wf(orid)
```

As you might have noticed, however, this function header isn't exactly like the previous two. This function is different from the others because it will take parameters (the variables input when calling the function) and will have an output that it returns.

The parameter we want it to take should be called "orid". This will allow us to specify the origin id of interest when we call the function so that it only gives us waveforms associated with that origin.

We'll just call the return value "out" to indicate that it is the output.

A fancy trick for functions written in MATLAB code is that one function can be specially designed to take different types and numbers of parameters. Even though we wrote the header for this function to indicate that we want to pass it the value of an "orid", we can allow the function to also work when no parameters are passed. In this case, we can make the function use the currently selected origin in the MatSeis window. This may sound difficult, but it only requires three lines of code:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check for any parameters.  If none
% are present, assume origin is already
% selected in MatSeis.
%
if nargin == 1
 origin('current', find(origin('orid') == orid));
end
```

The line `if nargin == 1` checks to see if the "n"umber of "arg"uments passed "in" (hence "nargin", which is an implicit MATLAB function) is equal to 1. If it is, then a parameter has been passed in and we call the 'origin' function with the parameters specified. The 'origin' function is another one of the five basic data types (like waveform) previously mentioned, and you can learn more about it by typing 'help origin' at the MATLAB command prompt.

The use of the origin function above isn't exactly trivial, so it probably warrants some explanation. Calling 'origin' with the parameter 'current' indicates that we are going to reset the index of the current origin to a new value. The new value should be specified in the next parameter. To decide what this new value should be, however, is not that simple. As you can see, the new value uses other function calls to get its value.

The expression `origin('orid')` returns an array holding all of the current origin ids loaded in MatSeis. Therefore, using an expression like `origin('orid') == orid` , as we do above, will check for equality among each entry of the array of origin ids and the value of the 'orid' variable that we input as a parameter. The result will be that the expression returns a Boolean array. That is, it returns an array of 1s and 0s where 1 indicates true (i.e. equality between that entry of the orid array and the input orid) and 0 indicates a false (i.e. inequality between that entry of the orid array and the input orid).

As an example, if we had given 'orid' the value of 6, and if `origin('orid')` returned an array containing the orids like [0 2 3 6 7], then `origin('orid') == orid` would give the array [0 0 0 1 0].

The find function, then, gives the nonzero indices of the array we just created . So, using the example above, calling 'find' on the array [0 0 0 1 0] would return a value of 4, the index of the nonzero term.

Putting this all together now, the line `origin('current', find(origin('orid') == orid))` will reset the index of the selected origin to the index of the origin whose id was input as a parameter (to 4 if we continue to use the example above).

Okay, that was a lot of explanation for a relatively simple concept. So, if you've got it down, let's move on.

Now we set the variable num_stachan to 14, the total number of characters that will be used (6 for the station name plus 8 for the channel name).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% From CSS3.0 sta = 6 chars,
% chan = 8 chars.
%
num_stachan = 14;
```

Now we are going to create an array holding the indices of the arrivals for this origin.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For arrivals, form arrays of stachan
% and of time.
%
arr_index = find(arrival('orid') == origin('orid', origin('current')));
```

The line `arrival('orid')` returns an array of the orids associated with the arrivals. The expression `origin('orid', origin('current'))` returns the 'orid' of the currently selected origin (remember we set this to be the orid that was passed in to the function). Comparing the array of arrival orids with the current orids using "==" will return a Boolean array with 1s representing arrivals that match the current orid and 0s for arrivals that that don't. The 'find' function, as before, will return the indices of the 1s that appear in the aforementioned Boolean array, and hence the indices of the arrivals associated with the current origin.

Now to find out how many arrivals there are, we simply need to find the size of the arr_index array we just created.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the number of arrivals
%
num_arriv = size(arr_index, 1);
```

If there are no arrivals, then we shouldn't be able to proceed any further. If this is the case, then we simply print out "No arrivals for this event" to the error dialog and then return to the calling function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bail if there are no arrivals...
%
if num_arriv == 0
  errordlg('No arrivals for this event');
  return;
end
```

If, on the other hand, there are some arrivals, then we will proceed to work with them. Our ultimate goal is to look for waveforms that have the same sta/chan as these arrivals *and* who have start/end times that contain the arrival times.

Let's begin by getting the sta/chan info for our associated arrivals. To do this, we need to set the arriv_stachan variable. We will concatenate (that is, put together) two separate strings so that they make up one string, and this single string will be given as the value of the 'arriv_stachan' variable.

The piece of code `arrival('sta',arr_index)` will return the station names at each arrival index, while `arrival('chan',arr_index)` will return the channel names at each arrival index.

When we then call the 'bpad' function on each of these strings along with an integer, each string is adjusted to the length specified by the integer. Therefore, the first string, the station name, will be either abbreviated or padded to 6 characters, and the second string, the channel name, will be abbreviated or padded to 8 characters. These will then be put together (either a space or the "…" will serve this purpose) forming an array of 14 character strings and assigned to the 'arrive_stachan' variable.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If there are arrivals, set the station
% channel.
%
arriv_stachan = [bpad(arrival('sta', arr_index), 6) ...
                 bpad(arrival('chan',arr_index), 8)];
```

We also need to get the times for these arrivals so we can look for waveforms that contain them. Calling the 'arrival' function with the parameters of 'time' and 'arr_index' will give an array holding the epoch times for each arrival index. These will then be assigned to the 'arriv_time' variable.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Also if there are arrivals, set the
% arrival time.
%
arriv_time = arrival('time', arr_index);
```

Now that we've taken care of the arrivals, we must do similar housekeeping for the actual waveforms with which these arrivals are associated. First, we get an array of all indices of waveforms loaded, then we get the size of this array to indicate how many waveforms there are.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Similarly for waveforms, get count of
% total number of waveforms in memory.
%
wf_index = waveform('index');
num_wf = size(wf_index, 1);
```

Now, like before, we set the name of the waveform station and channel by appending each name to the proper length and sticking them together.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set station and channel name.
%
wf_stachan = [bpad(waveform('sta', wf_index), 6) ...
              bpad(waveform('chan', wf_index), 8)];
```

When we set the time, however, we must do something a little bit different than before. Whereas the arrivals were only associated with a single moment in time, the waveforms have a beginning, a duration, and, therefore, an ending.

We get the start time as before. But to get the ending time, we have to add the duration to the start time. To get the duration, however, requires some math, for there is no implicit function of the 'waveform' datatype that gives the duration. We must instead divide the length of the waveform (the number of sampling points) by the sampling rate (number of sampling points per unit time). This will give us the total time (we can tell from the units of the measurements that we're doing this correctly,
i.e. # points / (# points / time) = time).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set the beginning and ending times
% for the waveforms.
%
wf_time1 = waveform('time', wf_index);
wf_time2 = wf_time1 + (waveform('length', wf_index)./waveform('samprate',
           wf_index));
```

Notice, however, the dot ('.') that appears immediately before the division sign. This is a special MATLAB notation that indicates the operation is to be done term by term as opposed to using regular matrix operations. For instance, if we were to use the following matrices, dot and regular operations would be executed as shown:

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> B = [5 6; 7 8]
B =
     5     6
     7     8

>> A./B
ans =
    0.2000    0.3333
    0.4286    0.5000
```

The dot operation of division divides the corresponding terms of the matrices. If we didn't use the dot operator, the division yields something entirely different. In fact, "A divided by B" without the dot operator is the same as "A times the inverse of B".

```
>> A/B
ans =
    3.0000   -2.0000
    2.0000   -1.0000

>> A*inv(B)
ans =
    3.0000   -2.0000
    2.0000   -1.0000
```

Using the dot operator in this way, the 'wf_time2' variable will be an array of properly calculated times.

Now we initialize an array call 'wf_vis' which will represent which waveforms should be left visible when we print them out in MatSeis. Calling the function `zeros(num_wf,1)` will create a column array containing only 0s that is the same size as the 'num_wf' variable. The 0s will indicate that the corresponding waveform should not be displayed.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize visibility array
%
wf_vis = zeros(num_wf,1);
```

Now we update the 'wf_vis' array using the information we've acquired in the previous steps to decide which waveforms will actually remain invisible. We need to insert a '1' into every index of a waveform that should be visible.

We do this using a for loop over the associated arrivals, checking each arrival's station and channel against the list of all waveform stations and channels and also checking whether the waveforms span the arrival time.

The whole code to determine which waveforms should be visible appears as follows. It's very complicated, but I'll explain it all in a minute.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For each arrival, turn on any
% corresponding waveform
%
for i = 1:num_arriv
   wf_vis = wf_vis | ...
            ((sum([wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) ...
            (arriv_time(i) >= wf_time1) & ...
            (arriv_time(i) <= wf_time2)]'))' == (num_stachan + 1));
end
```

Let's start with the basics. The 'wf_stachan' variable is a column of station and channel names. Depending on the actual station names and channels, a set of 4 waveforms might look something like this:

```
wf_stachan =

ADI   SHZ
ATZ   SHZ
CRI   SHZ
DOR   SHZ
```

The 'arriv_stachan' variable is a very similar column; it also contains station and channel names. The station and column names that it contains, however, can be of a different type, number, or order than those in the 'wf_stachan' column. It may look something like:

```
arriv_stachan =

ATZ   SHZ
CRI   SHZ
CRI   SHZ
MML   SHZ
ADI   SHZ
GLH   SHZ
ZNT   SHZ
```

Our job is to find the index of each 'wf_stachan' in the first column that matches one of the 'arriv_stachan' from the second column. These indices will represent all of the waveforms that should remain or become visible in MatSeis after our function is run.

As you may have noticed, the stachan "CRI SHZ" appears twice in the 'arriv_stachan' column, meaning that there are two arrivals associated with that station and channel. As you'll see shortly, this is inconsequential; a waveform will be made visible whether it has one arrival or hundreds.

Okay, so how are we actually going to compare these two columns? It seems like it should be simple. It isn't difficult to pick out the entries from the first column that have a match in the second column by simply eyeballing it. Unfortunately, there is no 'eyeball_it' function; it's going to be tough to actually write code to do this. We're going to have to get creative.

One possible solution is to take each entry of the second column individually and compare it to each entry of the first column. Let's give it a try.

Calling `arriv_stachan(i,:)` will give the i^{th} entry of the 'arriv_stachan' column. If we then call the MATLAB function 'repmat', we can replicate the matrix that represents that entry as many times as we want. The portion of the code that says `repmat(arriv_stachan(i,:), num_wf,1)` gives a column containing the i^{th} entry of 'arriv_stachan' as many times as there are waveforms. Continuing with the example above, calling `repmat(arriv_stachan(i,:), num_wf,1)` with i equal to 1, we would get…

```
repmat(arriv_stachan(1,:), num_wf,1) =

ATZ   SHZ
ATZ   SHZ
ATZ   SHZ
ATZ   SHZ
```

If we compare this to the 'wf_stachan' matrix using the code `wf_stachan == repmat(arriv_stachan(i,:), num_wf,1)` we'll end up getting a big matrix of 1s and 0s. There will be a 1 wherever the characters of the two matrices are the same and a 0 wherever they differ. In our example, we'll wind up with…

```
wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) =

1 0 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1
```

The line of all 1s represents the match, where every character is the same in both rows. This is not quite enough to give us what we're looking for, though. We also need to know that the arrivals we have fall somewhere on the correct time interval of the waveform. To do this, we'll add an extra column to the tail end of the matrix above representing whether the time interval is correct.

We do this with the portion of the code that says `(arriv_time(i) >= wf_time1) & (arriv_time(i) <= wf_time2)`. This code uses the logical operator "&", meaning "and". For the whole statement to be true and be assigned a 1 indicating that the arrival is in the correct time interval, the arrival must both come after the beginning of the waveform and before the end of the waveform. If either of these two criteria are false, then the whole statement is false.

```
arrival late enough and arrival early enough equals correct time interval
      1     &     0     =     0
      0     &     1     =     0
      1     &     1     =     1
      0     &     0     =     0
```

Again, we'll get a 1 if the time interval is correct and a 0 if not. So, assuming that the time interval is correct for this particular arrival, we'll get an extra column of 1s tacked on:

```
[wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) (arriv_time(i) >=
wf_time1) & ... (arriv_time(i) <= wf_time2)] =

1 0 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1
```

To decide whether or not we've found any equalities, we need to know whether or not we have any rows that are entirely full of 1s. Again, an 'eyeball_it' function would come in awfully handy, but we're going to have to figure out something else. Luckily, MATLAB provides a function called 'sum'. If we can sum up each row and find those that come out to 15 (the six 1s for that same station characters, the eight 1s for the same channel characters, and the extra 1 for the correct time interval), then we've found the row that has equality. We should write something like this:

```
((sum([wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) (arriv_time(i) >=
wf_time1) ... & (arriv_time(i) <= wf_time2)]'))' == (num_stachan + 1)) =

0
1
0
0
```

Finally, we've found a match. We now reset the 'wf_vis' variable to reflect the match that we've found. Remember that the 'wf_vis' variable is, before the code is run, a column of 0s. Every time we find a 1 representing a match, we need to put that into the 'wf_vis' variable in the correct spot. We do this using the logical operator "|", meaning "or". Our new 'wf_vis' should consist of 1s wherever the old 'wf_vis' had a 1 or wherever we found a match.

```
wf_vis = wf_vis | ...
         ((sum([wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) ...
         (arriv_time(i) >= wf_time1) & ...
         (arriv_time(i) <= wf_time2)]'))' == (num_stachan + 1));
```

The “or” operator would work as follows after going through the for loop one time.

```
old wf_vis or new match equals new wf_vis
      0     |     0     =     0
      0     |     1     =     1
      0     |     0     =     0
      0     |     0     =     0
```

But, like I’ve said, we’ve only gone through the for loop once. After we’ve traversed it for each arrival and the loop has finished, we should be left with…

```
wf_vis =

1
1
1
0
```

This means that the 1st, 2nd, and 3rd, but not the 4th waveform, waveforms have arrivals in the correct time interval associated with the given origin. The first 3 waveforms should be displayed.

Now, as the final step, we turn on all of the waveforms that should be visible so that they’ll appear in the MatSeis display. Then we zoom the MatSeis display to show all of the data displayed. These operations can be performed with the following two lines of code.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finally, turn on the waveforms
%
waveform('visibile', wf_index, wf_vis);
ms_zoom('data');
```

Finally, when we put this all together, we are left with a very useful piece of code.

```
function out = showassoc_wf(orid)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check for any parameters.  If none
% are present, assume origin is already
% selected in MatSeis.
%
if nargin == 1
  origin('current', find(origin('orid') == orid));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% From CSS3.0 sta = 6 chars,
% chan = 8 chars.
%
num_stachan = 14;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For arrivals, form arrays of stachan
```

```
% and of time.
%
arr_index = find(arrival('orid') == origin('orid', origin('current')));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the number of arrivals
%
num_arriv = size(arr_index, 1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bail if there are no arrivals...
%
if num_arriv == 0
  errordlg('No arrivals for this event');
  return;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If there are arrivals, set the station
% channel.
%
arriv_stachan = [bpad(arrival('sta', arr_index), 6) ...
                 bpad(arrival('chan',arr_index), 8)];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Also if there are arrivals, set the
% arrival time.
%
arriv_time = arrival('time', arr_index);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Similarly for waveforms, get count of
% total number of waveforms in memory.
%
wf_index = waveform('index');
num_wf = size(wf_index, 1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set station and channel name.
%
wf_stachan = [bpad( waveform('sta', wf_index), 6) ...
              bpad(waveform('chan', wf_index), 8)];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set the beginning and ending times
% for the waveforms.
%
wf_time1 = waveform('time', wf_index);
wf_time2 = wf_time1 + (waveform('length', wf_index)./waveform('samprate',
wf_index));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize visibility array
%
wf_vis = zeros(num_wf,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% For each arrival, turn on any
% corresponding waveform
%
for i = 1:num_arriv
  wf_vis = wf_vis | ...
           ((sum([wf_stachan == repmat(arriv_stachan(i,:), num_wf,1) ...
           (arriv_time(i) >= wf_time1) & ...
           (arriv_time(i) <= wf_time2)]'))' == (num_stachan + 1));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finally, turn on the waveforms
waveform('visibile', wf_index, wf_vis);
ms_zoom('data');
```

FUNCTION 4: AUTOMATICALLY MAKE WAVEFORM MEASUREMENTS FROM RECIPE FILE

As you have probably seen, these functions can become longer and more conceptually complicated, but the actual coding doesn't get much more difficult. If you are still doubtful of this, then let's see if I can't make my point a little more convincingly. It's time for a really long function…

We want the function to be able to automatically make measurements based on the specifications of the recipe file. Therefore, our function should take a recipe file as a parameter. We'll write the header as before.

```
function auto_meas(recipe_file);
```

Now if we're going to use the recipe file to decide what to do, we should probably figure out what it contains. To do this, we can read the file into a string matrix. We'll do this with the command 'readrect', a function provided by MATLAB. The code to do this looks as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read in recipe file
%
out = readrect(recipe_file, 77);
```

The line `out = readrect(recipe_file,77)` reads the recipe file into a matrix containing rows that are 77 characters apiece and assigns that matrix to the variable 'out'. The number 77 is not a magic one; the specifications of a recipe file designate that 77 characters are needed to fully describe all the fields and corresponding values of the file. For more information on the format of the recipe file, look at one of the example recipe files (e.g. SNL_Tool_Root\matseis\testing\matseis\northridge\nr.recipe or SNL_Tool_Root\matseis\testing\matseis\galilee\galilee.recipe).

Here, for instance, is the Northridge recipe file "nr.recipe":

```
STA    CHAN    PHASE     FLO    FHI   FPLS OFF1    GV1    OFF2    GV2   MEAS_TYPE   IC
NNA     BHZ     P       -1.0   -2.5    3     2.0   11.1     2.0   10.8   Zero-Peak    0
NNA     BHZ     P       -1.0   -2.5    3     2.0   11.1     2.0   10.8   RMS          0
KIV     BHZ     P       -1.0   -2.5    3     0.0   13.6     0.0   13.2   Zero-Peak    0
```

As you can see, parts of this file contain extraneous material, such as the header. We'll get rid of the header so that we're only working with the real content of the file.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get rid of header line
%
out = out(2:end,:);
```

This code reassigns the 'out' matrix to be the part of the original 'out' matrix from row 2 through the end with all columns. Basically, it removes the first line.

This snippet is an important piece of code because it introduces a new MATLAB syntax commonly known as "colon notation". Colon notation, designated as such because of its use of the colon, can often be used to replace a "for" loop. In fact, colon notation is greatly preferred to the for loop because it is much faster and less computationally expensive. In plain English, the colon can be thought of as meaning "for each". For instance, in the code above, the portion that says `2:end` which lies in the row position of the 'out' matrix means that we'll copy the rows for each row starting at the 2nd row and ending at the end. The lone colon in the column position of the 'out' matrix means that we'll copy the columns for each column.

The rest of the lines contain the valuable information that we want to extract and put into a certain structure. We set up this structure with the following piece of code.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup then populate recipe structure
%
recipe = struct('sta',[],'chan',[],'phase',[],...
                'flo',[],'fhi',[],'fpls',[],...
                'off1',[],'gv1',[],'off2',[],'gv2',[],...
                'meas_type',[],'ic',[]);
```

The 'struct' function, another MATLAB one, creates a structured array with the given fields and values. As you can see, the fields we are using are 'sta', 'chan', 'phase', 'flo', 'fhi', 'fpls', 'off1', 'gv1', 'off2', 'gv2', 'meas_type', and 'ic', each of which is associated with an empty value given by empty square brackets '[ ]'. This structured array is assigned to the variable 'recipe'.

Each line of the recipe file (except, of course, for the header which we already removed) contains a different "recipe". To treat each of these separately, we need to know how many of them there are. We can do this by simply calling the size function on the first column of the 'out' variable we created earlier to get the total number of rows. We do this as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get number of recipes
%
num_recipes = size(out,1);
```

Now we need to clear up the recipe structure so we can parse the recipe file and populate the recipe structure correctly.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the recipe matrix so we can
% repopulate with correct info.
%
recipe = recipe(ones(num_recipes,1));
```

This clears it out leaving only a column of 1s, one for each recipe in the file to preserve the size.

Now we actually go through and populate it.

We populate the structure for each recipe, so we actually will end up having multiple instances of the recipe variable. We do this using a for loop (using colon notation here would be far too messy).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Repopulate the recipe matrix
%
for i=1:num_recipes
  recipe(i).sta = deblank(out(i,1:5));
  recipe(i).chan = deblank(out(i,8:11));
  recipe(i).phase = deblank(out(i,15:18));
  recipe(i).flo = str2num(out(i,22:26));
  recipe(i).fhi = str2num(out(i,28:32));
  recipe(i).fpls = str2num(out(i,34:36));
  recipe(i).off1 = str2num(out(i,38:43));
  recipe(i).gv1 = str2num(out(i,45:49));
  recipe(i).off2 = str2num(out(i,51:56));
  recipe(i).gv2 = str2num(out(i,58:62));
  recipe(i).meas_type = deblank(out(i,64:73));
  recipe(i).ic = str2num(out(i,76:76));
end
```

The first line after the for loop begins will take the i^{th} recipe file and assign the first 5 characters of the i^{th} row of the 'out' matrix to its 'sta' field. The next line will take the i^{th} recipe file and assign the characters in lines 8-11 of the i^{th} row of the 'out' matrix to the 'chan' field. The remaining fields are populated similarly in the subsequent lines.

As a side note, the 'deblank' function is a MATLAB function that removes any extraneous empty spaces from the string, and the 'str2num' function is a MATLAB function that converts a string to a number. These are used to ensure proper formatting when the various fields are populated.

Now that we've organized all of the info from the recipe file and can access it, it's time to actually make the measurements. First we need a waveform to measure. Therefore, we need to know the origin and get the time associated with it. We get this information in what is, by now, hopefully a familiar manner.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now check the current origin and make
% measurements (if possible)
%
otime = origin('time','current');
```

We also need to know the station and channel of the waveforms to see if any match what was specified in the recipe file. Again, these can be retrieved quite easily and familiarly.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the station and channel of
% currenly selected waveform(s).
```

```
%
sta = waveform('sta');
chan = waveform('chan');
```

In case there are multiple waveforms selected to measure, we need to know how many there are. To figure this out, there are, again, no new tricks:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get number of waveforms selected.
%
num_wf = size(sta,1);
```

Now come the actual calculations. We need to perform them independently for each waveform selected, so we'll put the calculations into a for loop.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For each waveform, perform all
% measurements.
%
for i=1:num_wf
```

Because each recipe from the recipe file is associated with a given station and channel, we need to make certain that these (or at least some of these) match those actually loaded and selected in MatSeis. To do this, we need to use the MATLAB string comparator function called 'strcmp'.

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Check for a recipe for this waveform
  % (sta & chan).
  %
  index = find(strcmp({recipe(:).sta}, deblank(sta(i,:))) & ...
               strcmp({recipe(:).chan}, deblank(chan(i,:))));
```

The line `strcmp({recipe(:).sta}, deblank(sta(i,:)))` compares the names of the stations for each recipe and the current waveform being processed. The line `strcmp({recipe(:).chan}, deblank(chan (i,:)))` does the same for the names of the channels for each recipe and the current waveform being processed. If both of these match (hence the '&') then the 'find' function will return the index of the match and thus give the index of the recipe that corresponds to the selected waveform.

If there is at least one match, then we can proceed with the measurements. If there is no match, then there is nothing we can do. We check this condition in the same manner as previous times.

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If there is at least one match
  % then we proceed.
  %
  if ~isempty(index)
```

All we've checked for at this point, however, is if at least one recipe matches. There could have been more than one match. We need to get the total number of matches so that we can make measurements based on each recipe individually.

To do this, we again use a for loop. It's pretty much the same story as before:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get total number of matches.  Perform
% measurements for each recipe file
% individually.
%
num_recipe = length(index);
for j=1:num_recipe
```

Once we've got matching stations and channels for a recipe and a waveform, we should also check that the time intervals of these coincide. We must first calculate the time intervals for the recipe and for the actual waveform.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Calculate the time intervals
    %
    meas_time1 = otime + recipe(index(j)).off1 + ...
            deg2km(waveform('distance',i)) / recipe(index(j)).gv1;
    meas_time2 = otime + recipe(index(j)).off2 + ...
            deg2km(waveform('distance',i)) / recipe(index(j)).gv2;
    wfm_time1 = waveform('time',i);
    wfm_time2 = wfm_time1 + waveform('length',i)./waveform('samprate',i);
```

The first measured time is calculated by starting with the origin time 'otime', adding the offset time 'off1', and adding the time computed by dividing the distance (which the MatSeis function 'deg2km' converts from epicentral degrees to kilometers) from the origin by the group velocity 'gv1'. The second measured time is calculated similarly.

The time span of the waveforms themselves is easy to calculate. We do it as we did previously, first getting the start time by calling `wfm_time1 = waveform('time',i)` and then getting the stop time by adding the duration (which we must calculate) to the start time with the code `wfm_time2 = wfm_time1 + waveform('length',i)./waveform('samprate',i).`

Now we check to make the time intervals actually overlap. We need the waveform to entirely contain the measured time interval specified by the recipe.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Check to see if waveform is in right
    % time interval.  Note that this check
    % is conservative; it must have the full
    % waveform available!!
    %
    if wfm_time1 < meas_time1 & wfm_time2 > meas_time2
```

Once it is determined that the time intervals do, in fact, correctly overlap, then we can move on. We're getting close to actually making the measurement, so maybe we should give the user a heads up that that's what we're going to do. To do this, we print out to the display what station, channel, and measurement type we're making.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Print out a display.
%
disp(['making measurement for ' recipe(index(j)).sta ...
     recipe(index(j)).chan ' ' recipe(index(j)).phase ...
     ' ' recipe(index(j)).meas_type]);
```

Now, because our convention is to link measurements to arrivals, we should check if there are any arrivals associated with the particular origin, station, and phase.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check if there is an arrival for
% this phase.
%
ar_index = find(arrival('orid') == origin('orid', 'current') & ...
     compstr(arrival('sta'), recipe(index(j)).sta) & ...
     compstr(arrival('phase'), recipe(index(j)).phase));
```

If there is no existing arrival that has already been picked for the given phase and origin, then we need to pick one automatically. This can be done as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If no arrival exists, pick one.
%
if isempty(ar_index)
  ar_index = arrival('create',1);
  arrival('orid', ar_index, origin('orid', 'current'));
  arrival('sta', ar_index, recipe(index(j)).sta);
  arrival('chan', ar_index, recipe(index(j)).chan);
  arrival('phase', ar_index, recipe(index(j)).phase);
  arrival('time', ar_index, meas_time1);
end
```

For any of the parameters that don't already exist, assignments are made based on the values given in the recipe file. As you can see in the code above, the arrival gets an orid, a station, a channel, a phase, and a time. As the only purpose of this arrival is to provide something to tie measurements to, we will just set the time to be that determined by the first group velocity.

Now, for a little housekeeping, we should make certain that we don't accidentally have any duplicate arrivals.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make sure there are no duplicates.
%
ar_index = ar_index(1);
```

Once we've done that, there's nothing to stop us from actually working with the data. First, we read it into a variable that we'll call (what else?) 'data'. This can be done very easily using the prewritten waveform function that we talked about in previous sections. While we're at it, we'll also get the total number of data points for future reference.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read in the data and get total amount
% of data.
%
data = waveform('data',i);
n = length(data);
```

Now we'll get the time at each data point for the current waveform. We do this by adding the initial time to the duration, up to the given point, of the time since the initial one. We can do this much like before.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the time at each data point.
%
time = waveform('time',i) + (0:n-1)'/waveform('samprate',i);
```

Before we measure the waveform, we need to filter it. The recipe file actually specifies several of the filtering criteria, so we'll use them here. First, we create an empty variable called 'cutoff' that will hold any of the filtering criteria we add. If no filtering is done, it will remain empty.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start with no filter.
%
cutoff = [];
```

We then use several embedded 'if' statements to decide what filtering to apply. The first 'if' decides whether there is a low end cutoff frequency to filter. Inside of this, we decide with another 'if' whether there is a high end cutoff frequency to filter. If there is, then we have both a low and a high and therefore a general "band" of frequencies in which to stay. If there is a low but no high, then we have a low band cutoff. If there is a high but no low, then we have a high band cutoff. If there are neither, then there is no filtering to be done.

Our code to determine this functionality should appear as follows:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decide how to filter the waveform
% according to the recipe file.
%
if recipe(index(j)).flo > 0
  if recipe(index(j)).fhi > 0
    cutoff = [recipe(index(j)).flo recipe(index(j)).fhi];
    band = 'band';
  else
    cutoff = [recipe(index(j)).flo];
    band = 'low';
  end
else
  if recipe(index(j)).fhi > 0
    cutoff = [recipe(index(j)).fhi];
    band = 'high';
  end
```

```
        end
```

We check the values specified in the recipe file for the frequency cutoffs, and if they are valid we set them as the values of the 'cutoff' variable and note which type of filter they specify with the 'band' variable.

If some sort of filtering is required, we must perform it now. The 'cutoff' variable holds all of the information describing what kind of filtering is needed.

So, assuming that there are filtering values specified and the 'cutoff' variable isn't empty, we apply filtering to the waveform.

The variable name 'nyquist' is a signal processing term related to the sampling frequency of data used to represent the waveform. It turns out that the waveform can only be accurately recreated if it has no frequencies higher than one-half the sampling frequency. We call this cutoff frequency, one-half the sampling rate, the nyquist frequency.

Because of MATLAB filtering conventions, we then calculate the normalized cutoff frequency by dividing the cutoff frequency (or frequencies) specified in the recipe file by the nyquist frequency. The code to do all of this appears as follows:

```
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Get nyquist frequency and calculate
        % the normalized cutoff frequency.
        %
        if ~isempty(cutoff)
          nyquist = waveform('samprate',i)/2.;
          Wn = cutoff/nyquist;
```

Once we have this information, we can use the implicit MATLAB function 'filterd' to generate the coefficients for the filter we specified. We do this with the line of code `[b, a] = filterd('Butterworth', recipe(index(j)).fpls, Wn, band)`. This will use a Butterworth filter of order `recipe(index(j)).fpls` with normalized cutoff frequency `Wn` on the specified band.

We then use these coefficients to reconstruct the waveform with the line of code `data = filtfilt(b, a, data)`, which uses another MATLAB function called 'filtfilt' to apply a "zero-phase forward and reverse" filter.

```
          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
          % Filter the waveforms.
          %
          [b, a] = filterd('Butterworth', recipe(index(j)).fpls, Wn, band);
          data = filtfilt(b, a, data);
      end
```

Now that the filtering is done, we can finally take the measurements we set out to make so long ago. To do this is simple; we just call the 'measure' function with the specified criteria. BUT WAIT, it turns out that there isn't a 'measure' function! No such function exists! So what do we do now?

Well, doesn't it seem like it would be useful if there was, in fact, a 'measure' function? Then we could do exactly what we tried to do a minute ago… simply call it with the appropriate criteria. Well, as long as we're writing one function, why not write two?

Why not indeed. In fact, this is exactly what we'll do. We'll write a *sub*function called 'measure' that will allow us to quickly and easily measure the waveform using whatever criteria we need it to accept. Once it's written, we can use it wherever we want.

So let's pretend that such a function exists, call it now like we normally would, and actually write it later.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make measurements
%
[meas] = measure(data, time, meas_time1, meas_time2,...
                 recipe(index(j)).meas_type, i);
```

We call the function with the parameters of 'data', 'time', 'meas_time1', 'meas_time2', 'meas_type', and 'i', each of which will be used in the calculation of the waveform's measurement. The results are returned in the structure 'meas'.

Now we need to set the amplitude and period of the arrival associated with the waveform we measured so it contains the updated measurements. We'll also retrieve the arrival id of this arrival so we can use it when we record the measurement. We do this with the following code.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create a new arrival with the given
% measurements.
%
arrival('amp', ar_index, meas.amp);
arrival('per', ar_index, meas.per);
arid = arrival('arid', ar_index);
```

Now we actually have to set up a new measurement entry so we can record all of the information that we just acquired with our 'measure' function. First we'll get the number of measurements that have already been made and advance that by 1 to get the index for the new measurement we plan to store. The first time through, if no measurements are already stored in MatSeis, measurement('n') will be 0 so our 'num_amps' index will be 1. Then we allocate a new blank (null) measurement entry, which will correspond to the index 'num_amps' and call it 'temp' because it will only temporarily hold our data.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up a new measurement entry.
%
```

```
        num_amps = measurement('n') + 1;
        temp = measurement('null',1);
```

Now we populate our 'measurement' entry with all of the values of data that we've got. The first call of the 'measurement' function, the line `measurement('data', num_amps, temp)`, sets the 'num_amps' object to be our empty 'temp' object. Therefore, in each subsequent call, we'll be populating the various values of a blank temporary measurement object. Basically, it allows us to start from scratch and create an entirely new instance of our 'measurement' object.

So let's start filling in our blank object. We start by setting the 'arid' field with the arrival id we just retrieved a few lines ago. We do this using the MatSeis function 'measurement' by calling it as follows: `measurement('arid', num_amps, arid)`.

For the 'sta', 'chan', 'phase', and 'amptype' fields associated with the measurement, we simply use the 'sta', 'chan', 'phase', and 'amptype' names dictated in the recipe file. We call the 'cellstr' function on these names to solidify each one as a separate string object rather than as an array of characters. Each of these is set to its respective field in the same manner as the 'arid'.

Following this are the 'amptime', 'amp', and 'per' fields. These fields are populated using the information garnered from our 'measure' function. We set the 'amptime' to the time associated with the 'meas' variable we returned from the measure function. We reference this value by calling `meas.time` and set it to the 'measurement' object with the line `measurement('amptime', num_amps, meas.time)`. The amplitude and period are accessed and set similarly.

Lastly, we assign the 'start_time' and 'duration' fields using the 'meas_time1' and 'meas_time2' variables that we calculated earlier.

Putting all of these field assignments together and then wrapping up our function with a series of 'end' statements, our code should look as follows.

```
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Populate measurement fields.
        %
        measurement('data', num_amps, temp);
        measurement('arid', num_amps, arid);
        measurement('sta', num_amps, cellstr(recipe(index(j)).sta));
        measurement('chan', num_amps, cellstr(recipe(index(j)).chan));
        measurement('phase', num_amps, cellstr(recipe(index(j)).phase));
        measurement('amptype', num_amps,cellstr(recipe(index(j)).meas_type));
        measurement('amptime', num_amps, meas.time);
        measurement('amp', num_amps, meas.amp);
        measurement('per', num_amps, meas.per);
        measurement('start_time', num_amps, meas_time1);
        measurement('duration', num_amps, meas_time2 - meas_time1);
      end
    end
  end
end
```

So now that we're done with our main function, it's time to introduce another new topic in MatSeis function writing… the subfunction. Subfunctions, also known as helper functions, are written within the same M file as the main function, and they use the same syntax. Their purpose, however, is to "help" the main function. They cannot, therefore, be called outside of file that contains the main function. Basically, the only difference between a main function and a subfunction is that a subfunction must appear after the main function. If there are multiple subfunctions, they may appear in any order, but they must all appear after the main function.

Subfunctions are often used if there is a particular task that is to be repeated multiple times within the main function. It makes the code more readable, more understandable, and it's easier to write.

The subfunction that we're going to write will allow us to measure the waveform. We'll call it 'measure' and give it a header. However, we have to remember that when we wrote the function call before, we passed in a handful of parameters. So now, as we're actually writing the body of the function, we need to make sure our function header specifies that it should accept those same parameters. In this case, we need it to take the parameters of the 'data' we want to measure, the 'time', the type of measurement we want to make, and the index of the waveform to which all of the other parameters belong.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                                SUBFUNCTION                              %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [meas] = measure(data, time, time1, time2, meas_type, wfm_index);
```

We'll start off by setting our output variable, 'meas', to be an empty vector.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the meas variable.
%
meas = [];
```

Then we'll use the parameters passed in to construct a set of data that is consistent with what we want to measure. First, we construct a Boolean array called 'range' that signifies which of the times in the 'time' parameter fall within the time interval specified. We then use the piece of code `data = data(range)` to truncate the 'data' variable so it doesn't contain any of the extraneous data falling outside of the interval we're interested in. We also update the 'time' variable in a similar manner so that it still corresponds to the data correctly. The code for this appears as follows:

```
range =  time >= time1 & time <= time2;
data = data(range);
time = time(range);
```

Once we've whittled down the data we want to measure, there are three possible cases for the type of waveform measurement we want to make: peak to peak, zero to peak, and RMS (root mean square). The 'meas_type' field in the recipe file should specify which of these types of measurements is to be taken for the data.

We decide which case to use with a 'switch' statement. This is similar to an 'if' statement, but actually uses specific cases to decide what should be done. In our case, we'll "switch" using the string representing the measurement type. However, because the switch statement is case sensitive, we must make sure to match the capitalization of the strings on which we are switching. To do this, we can use the MATLAB function 'lower'. This will convert any string to all lower case letters so that we know what to expect when the switch statement compares the strings. The code looks like this:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decide which type of measurement to
% make.
%
switch(lower(meas_type))
```

And we follow this immediately with the possible cases. Each case gets its own block of code to determine the specific actions to take in the given case. We'll write the first of these cases now, the case when our measurement type is 'peak-peak'.

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If peak to peak is specified, execute
  % this code.
  case 'peak-peak'
    [ymin, imin] = min(data);
    [ymax, imax] = max(data);
    meas.time = min(time(imin), time(imax));
    meas.per  = 2.*abs(time(imin) - time(imax));
    meas.amp  = ymax - ymin;
    meas
```

For a peak-peak measurement, the amplitude is measured as the difference between the highest and lowest points on the waveform, the period is double the time difference between these two points, and the time is the time associated with the beginning of the period being measured. We can find these high and low points as well as their indices very easily using MATLAB's 'max' and 'min' functions. The line `[ymin, imin] = min(data)` will set 'ymin' to the minimum value of the data and 'imin' to the index of that value. The 'max' function works similarly.

We assign the value to the 'time' field with the line `meas.time = min(time(imin), time(imax))`, we assign the value to the 'per' field with the line `meas.per = 2.*abs(time(imin) - time(imax))`, and we assign the value to the 'amp' field with the line `meas.amp = ymax – ymin`.

Next, we'll write the block of code used when a 'zero-peak' measurement is specified. In this case, all of the measurements except that of amplitude are exactly the same as in a 'peak-peak' measurement. The amplitude, however, is calculated differently. It picks the greater of the absolute values of the max and min values, then assigns it a positive or negative sign depending on which of the two values was used. This is executed with the line of code `meas.amp = sign(ymin + ymax)*max(abs(ymin), abs(ymax))`. This is followed by another 'if' statement for any instrument corrections that need to be made. The code for the 'zero-peak' case appears as follows:

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If zero to peak is specified, execute
  % this code.
  %
  case 'zero-peak'
    [ymin, imin] = min(data);
    [ymax, imax] = max(data);
    meas.time = min(time(imin), time(imax));
    meas.per  = 2.*abs(time(imin) - time(imax));
    meas.amp  = sign(ymin + ymax)*max(abs(ymin), abs(ymax));
    meas
```

For the last case, a root mean square measurement, we use the square root of the average of the squares of the all the data measurements to calculate the amplitude. We get this value using the line `rms = sqrt(sum(data.^2)/length(data))`. For our measurement time, we use the variable 'time1' because it is time associated with the beginning of our measurement. Our "period" will be 'time2' minus 'time1', the span of the whole set of data. The code for the 'rms' case should look something like this:

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If root mean square is specified,
  % execute this code.
  %
  case 'rms'
    rms = sqrt(sum(data.^2)/length(data));
    meas.time = time1;
    meas.per  = time2 - time1;
    meas.amp  = rms;
    meas
```

Once we've written the code for all of the cases we would expect to encounter, we should write one last statement to deal with any unexpected measurement types. To do this, we use the keyword 'otherwise' to take care of all other cases. In our 'otherwise' case, we print out a warning to inform the user that an unknown measurement type was requested and tell them what the measurement type was called. Once we do this, we can end our switch statement and, with it, our 'measure' function. We do this all with the following line of code:

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If none of previous cases, execute
  % this code.
  %
  otherwise warning(['unknown measurement type: ' meas_type]);
end
```

And that's it. Our 'auto_meas' function (replete with 'measure' subfunction) is complete. Putting all of our code together, the final function should look something like this.

```
function auto_meas(recipe_file);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read in recipe file
```

```
%
out = readrect(recipe_file, 77);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get rid of header line
%
out = out(2:end,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup then populate recipe structure
%
recipe = struct('sta',[],'chan',[],'phase',[],...
                'flo',[],'fhi',[],'fpls',[],...
                'off1',[],'gv1',[],'off2',[],'gv2',[],...
                'meas_type',[],'ic',[]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get number of recipes
%
num_recipes = size(out,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the recipe matrix so we can
% repopulate with correct info.
%
recipe = recipe(ones(num_recipes,1));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Repopulate the recipe matrix
%
for i=1:num_recipes
  recipe(i).sta = deblank(out(i,1:5));
  recipe(i).chan = deblank(out(i,8:11));
  recipe(i).phase = deblank(out(i,15:18));
  recipe(i).flo = str2num(out(i,22:26));
  recipe(i).fhi = str2num(out(i,28:32));
  recipe(i).fpls = str2num(out(i,34:36));
  recipe(i).off1 = str2num(out(i,38:43));
  recipe(i).gv1 = str2num(out(i,45:49));
  recipe(i).off2 = str2num(out(i,51:56));
  recipe(i).gv2 = str2num(out(i,58:62));
  recipe(i).meas_type = deblank(out(i,64:73));
  recipe(i).ic = str2num(out(i,76:76));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now check the current origin and make
% measurements (if possible)
%
otime = origin('time','current');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the station and channel of
% currenly selected waveform(s).
%
sta = waveform('sta');
chan = waveform('chan');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get number of waveforms selected.
%
num_wf = size(sta,1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For each waveform, perform all
% measurements.
%
for i=1:num_wf

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Check for a recipe for this waveform
  % (sta & chan).
  %
  index = find(strcmp({recipe(:).sta}, deblank(sta(i,:))) & ...
               strcmp({recipe(:).chan}, deblank(chan(i,:))));

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If there is at least one match
  % then we proceed.
  %
  if ~isempty(index)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Get total number of matches.  Perform
    % measurements for each recipe file
    % individually.
    %
    num_recipe = length(index);
    for j=1:num_recipe

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      % Calculate the time intervals
      %
      meas_time1 = otime + recipe(index(j)).off1 + ...
              deg2km(waveform('distance',i)) / recipe(index(j)).gv1;
      meas_time2 = otime + recipe(index(j)).off2 + ...
              deg2km(waveform('distance',i)) / recipe(index(j)).gv2;
      wfm_time1 = waveform('time',i);
      wfm_time2 = wfm_time1 + waveform('length',i)./waveform('samprate',i);

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      % Check to see if waveform is in right
      % time interval.  Note that this check
      % is conservative; it must have the full
      % waveform available!!
      %
      if wfm_time1 < meas_time1 & wfm_time2 > meas_time2

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Print out a display.
        %
        disp(['making measurement for ' recipe(index(j)).sta ...
             recipe(index(j)).chan ' ' recipe(index(j)).phase ...
             ' ' recipe(index(j)).meas_type]);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check if there is an arrival for
% this phase.
%
ar_index = find(arrival('orid') == origin('orid', 'current') & ...
     compstr(arrival('sta'), recipe(index(j)).sta) & ...
     compstr(arrival('phase'), recipe(index(j)).phase));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If no arrival exists, pick one.
%
if isempty(ar_index)
  ar_index = arrival('create',1);
  arrival('orid', ar_index, origin('orid', 'current'));
  arrival('sta', ar_index, recipe(index(j)).sta);
  arrival('chan', ar_index, recipe(index(j)).chan);
  arrival('phase', ar_index, recipe(index(j)).phase);
  arrival('time', ar_index, meas_time1);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Make sure there are no duplicates.
%
ar_index = ar_index(1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Read in the data and get total amount
% of data.
%
data = waveform('data',i);
n = length(data);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get the time at each data point.
%
time = waveform('time',i) + (0:n-1)'/waveform('samprate',i);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Start with no filter.
%
cutoff = [];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decide how to filter the waveform
% according to the recipe file.
%
if recipe(index(j)).flo > 0
  if recipe(index(j)).fhi > 0
    cutoff = [recipe(index(j)).flo recipe(index(j)).fhi];
    band = 'band';
  else
    cutoff = [recipe(index(j)).flo];
    band = 'low';
  end
else
  if recipe(index(j)).fhi > 0
```

```
            cutoff = [recipe(index(j)).fhi];
            band = 'high';
          end
        end

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Get nyquist frequency and calculate
        % the normalized cutoff frequency.
        %
        if ~isempty(cutoff)
          nyquist = waveform('samprate',i)/2.;
          Wn = cutoff/nyquist;

          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
          % Filter the waveforms.
          %
          [b, a] = filterd('Butterworth', recipe(index(j)).fpls, Wn, band);
          data = filtfilt(b, a, data);
        end

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Make measurements
        %
        [meas] = measure(data, time, meas_time1, meas_time2,...
                         recipe(index(j)).meas_type, i);

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Create a new arrival with the given
        % measurements.
        %
        arrival('amp', ar_index, meas.amp);
        arrival('per', ar_index, meas.per);
        arid = arrival('arid', ar_index);

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Set up a new measurement entry.
        %
        num_amps = measurement('n');
        temp = measurement('null',1);
        num_amps = num_amps + 1;

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Populate measurement fields.
        %
        measurement('data', num_amps, temp);
        measurement('arid', num_amps, arid);
        measurement('sta', num_amps, cellstr(recipe(index(j)).sta));
        measurement('chan', num_amps, cellstr(recipe(index(j)).chan));
        measurement('phase', num_amps, cellstr(recipe(index(j)).phase));
        measurement('amptype', num_amps,
cellstr(recipe(index(j)).meas_type));
        measurement('amptime', num_amps, meas.time);
        measurement('amp', num_amps, meas.amp);
        measurement('per', num_amps, meas.per);
        measurement('start_time', num_amps, meas_time1);
        measurement('duration', num_amps, meas_time2 - meas_time1);
      end
```

```
    end
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                              SUBFUNCTION                                 %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [meas] = measure(data, time, time1, time2, meas_type, wfm_index);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear the meas variable.
%
meas = [];
range =  time >= time1 & time <= time2;
data = data(range);
time = time(range);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Decide which type of measurement to
% make.
%
switch(lower(meas_type))

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If peak to peak is specified, execute
  % this code.
  case 'peak-peak'
    [ymin, imin] = min(data);
    [ymax, imax] = max(data);
    meas.time = min(time(imin), time(imax));
    meas.per  = 2.*abs(time(imin) - time(imax));
    meas.amp  = ymax - ymin;
    meas

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If zero to peak is specified, execute
  % this code.
  %
  case 'zero-peak'
    [ymin, imin] = min(data);
    [ymax, imax] = max(data);
    meas.time = min(time(imin), time(imax));
    meas.per  = 2.*abs(time(imin) - time(imax));
    meas.amp  = sign(ymin + ymax)*max(abs(ymin), abs(ymax));
    meas

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If root mean square is specified,
  % execute this code.
  %
  case 'rms'
    rms = sqrt(sum(data.^2)/length(data));
    meas.time = time1;
    meas.per  = time2 - time1;
    meas.amp  = rms;
    meas
```

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % If none of previous cases, execute
  % this code.
  %
  otherwise warning(['unknown measurement type: ' meas_type]);
end
```

CONCLUSIONS

In this MatSeis Developer's Guide, we have shown the reader how to use MATLAB and MatSeis to develop four seismic algorithms of increasing complexity, which we hope will provide them with enough knowledge to move on to developing MatSeis algorithms on their own. The examples showed how to make use of many useful MATLAB utilities and more importantly, introduced the MatSeis functions that provide access to our five core data-types: waveforms, origins, arrivals, travel times, and measurements. Mastering the use of these functions is the key to developing algorithms using MatSeis.

REFERENCES

Harris, M. and C. Young, 1997. MatSeis: a seismic GUI and tool-box for MATLAB, Seism. Res. Lett., 68, 267-269.

DISTRIBUTION

1 Leslie Casey
NNSA Office of Nonproliferation Research and Development/NA-22
1000 Independence Avenue SW
Washington, DC 20585

1 Mark Woods
Air Force Technical Applications Center/TTR
1030 S. Highway A1A
Patrick AFB, FL 32925-3002

1 Dean Clauter
Air Force Technical Applications Center/TTR
1030 S. Highway A1A
Patrick AFB, FL 32925-3002

1 John Dwyer
Air Force Technical Applications Center/TTR
1030 S. Highway A1A
Patrick AFB, FL 32925-3002

1 Jorge Roman-Nieves
Air Force Technical Applications Center/TTR
1030 South Highway A1A
Patrick AFB, FL 32925-3002

1 Jeff Miller
Air Force Technical Applications Center/TTR
1030 South Highway A1A
Patrick AFB, FL 32925-3002

1 Jon Creasey
Air Force Technical Applications Center/TTR
1030 S. Highway A1A
Patrick AFB, FL 32925-3002

1 Gordon Kraft
Air Force Technical Applications Center/TNDC/QTSI
Suite 514
1980 N. Atlantic Avenue
Cocoa Beach, FL 32931

1 Michael Begnaud
Los Alamos National Laboratory
MS F6665
P.O. Box 1663
Los Alamos, NM 87545

1 W. Scott Phillips
Los Alamos National Laboratory
MS D408
EES-11 P.O. Box 1663
Los Alamos, NM 87545

1 George Randall
Los Alamos National Laboratory
MS D408
EES-11 P.O. Box 1663
Los Alamos, NM 87545

1 Charlotte Rowe
Los Alamos National Laboratory
MS D408
EES-11 P.O. Box 1663
Los Alamos, NM 87545

1 Rodney Whitaker
Los Alamos National Laboratory
MS J577
EES-2 P.O. Box 1663
Los Alamos, NM 87545

1 Leigh House
NNSA Office of Nonproliferation Research & Development/NA-22/LANL
GH-068
1000 Independence Ave SW
Washington, DC 20585

1 Samuel Heller
Lawrence Livermore National Laboratory
L-205
P.O. Box 808
Livermore, CA 94551

1 Doug Dodge
Lawrence Livermore National Laboratory
MS L202
PO Box 808
Livermore, CA 94551-0808

1 Dale Anderson
Pacific Northwest National Laboratory
MS K5-12
P.O. Box 999
Richland, WA 99352-0999

1 Rick Schult
Air Force Research Laboratory/VSBYE
29 Randolph Road
Hanscom AFB, MA 01731-3010

1 Robert Shumway
University of California Davis
Division of Statistics 1 Shields Avenue
Davis, CA 95616-8500

1	MS0401	Jeff Hampton	05533
1	MS0401	Jake Jones	05533
1	MS0401	John Merchant	05533
1	MS0401	Chris Young	05533
1	MS0404	Eric Chael	05736
1	MS0404	Mark Harris	05736
1	MS0404	Darren Hart	05736
1	MS0750	Greg Elbring	06314
1	MS0750	Neill Symons	06314
2	MS9018	Central Technical Files	8944
2	MS0899	Technical Library	4536