TECHNICAL NOTE

Anritsu

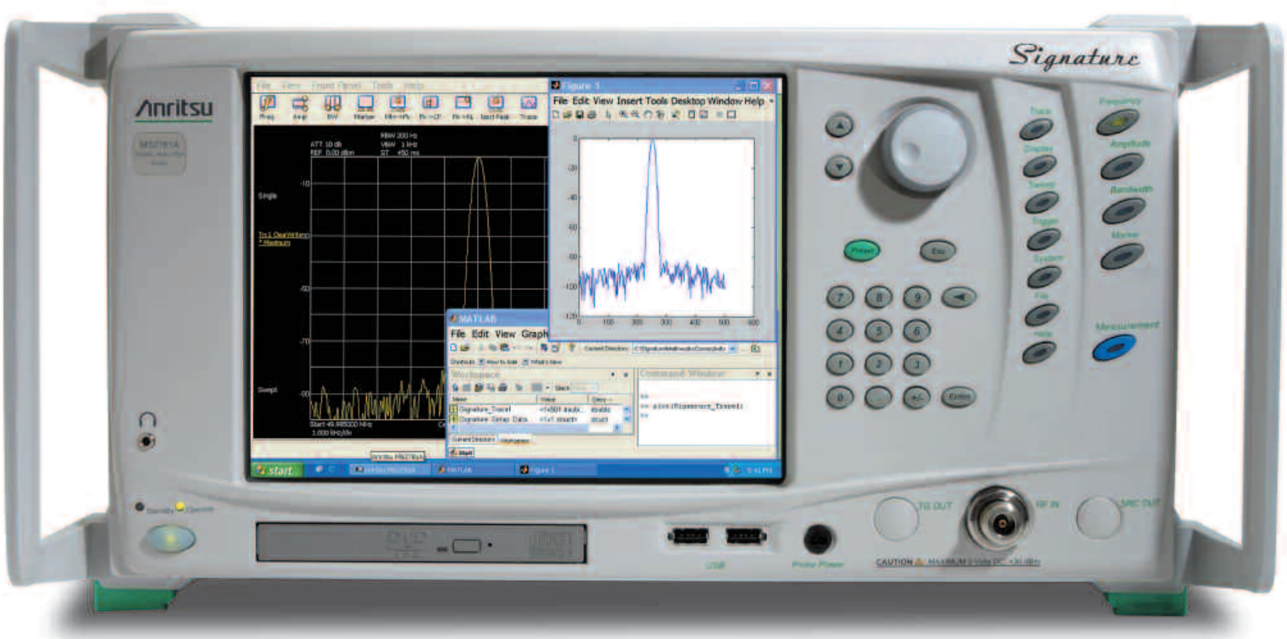
Custom Measurements and Analysis Using MATLAB® on Signature™

Signature™ MS2781A Signal Analyzer

Introduction

Signature is a combined high performance Spectrum Analyzer for characterizing RF signals and a high performance Vector Signal Analyzer for characterizing digitally modulated signals. Signature expands the ability to analyze RF signals by offering seamless connectivity with MATLAB® and Simulink® from The MathWorks. Engineers can view measurement results through custom MATLAB and Simulink analysis giving exceptional insight into the performance of new designs.

This technical note describes how to make this connection, and uses a number of examples to illustrate the power of this combination. Simulink and MATLAB options, such as the Signal Processing Toolbox, are also illustrated. If you are not already using MATLAB, you can find out more information from the MathWorks web site, at: www.mathworks.com. A limited-time trial version of MATLAB and other MathWorks products is also available to Signature users. Go to the following web address to find details about this trial offer: www.mathworks.com/anritsu.



Version 2.0, September 2005

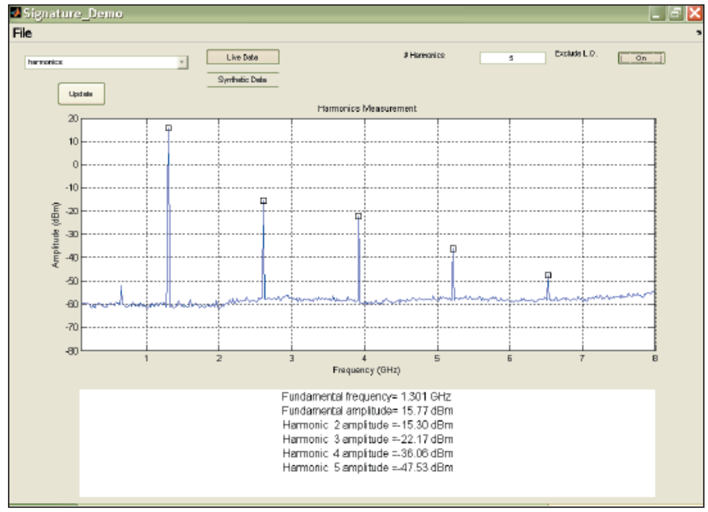
Table of Contents

Introduction . . . 1
MATLAB Analysis Examples . . . 3
Installing MATLAB on Signature . . . 5
Configuring Signature to use MATLAB . . . 5
The MATLAB Desktop Window . . . 5
Getting Setup Information from Signature into MATLAB . . . 6
Getting Data from Signature into MATLAB . . . 6
Viewing the trace values . . . 6
Drawing a Signature trace in MATLAB . . . 7
Plot . . . 7
Loops . . . 8
Timers . . . 9
Synchronization . . . 10
Trace Averaging with Handshaking . . . 11
Storing Multiple Traces with Handshaking . . . 12
Manual Sweep with Handshaking . . . 12
Timers with Handshaking . . . 13
Zero-span traces . . . 14
Modulation Measurements . . . 15
IQ Vectors . . . 16
Plotting IQ Vectors . . . 17
Plotting the Magnitude of IQ vectors . . . 17
FFT of IQ vectors . . . 17
FFT with Windowing . . . 18
I and Q magnitudes . . . 18
I and Q Polar plot . . . 19
Saving captured IQ vectors to the Anritsu MG3700A Vector Signal Generator . . . 19
Example Applications . . . 20
Spectral Measurements . . . 20
Channel Power . . . 20
Channel Power with Filtering . . . 21
Channel Power Function with Optional Filtering . . . 22
Adjacent Channel Power (ACP) . . . 23
Noise Compensation . . . 24
Plotting a Trace and Measuring ACP with Noise Compensation . . . 25
Multi-Carrier Power . . . 26
Harmonics . . . 28
Occupied Bandwidth . . . 29
Power Spectral Density . . . 31
IQ Measurements . . . 32
Frequency versus Time . . . 32
CCDF . . . 33
Spectrograms . . . 34
Specgram function . . . 34
Labeling the Spectrogram Axes . . . 35
MATLAB Spectrogram Demo . . . 35
Building your own Spectrogram . . . 35
Spectrograms from IQ vectors vs. from Traces . . . 35
Using Simulink . . . 36
FSK Demodulation . . . 36
“To Frequency” Block . . . 37
“Measure FSK” block . . . 37
Getting MATLAB data into Simulink . . . 38
Measurement Results . . . 38
Instrument Control . . . 39
Controlling Signature through Web Services . . . 39
GPIB Control of Other Instruments from MATLAB . . . 41
Example of Controlling Instruments—Measuring ACPR versus Power . . . 41
MATLAB Demonstration . . . 42
Using the MATLAB Demonstration . . . 42
How to get support . . . 43
Conclusion . . . 43
References . . . 43

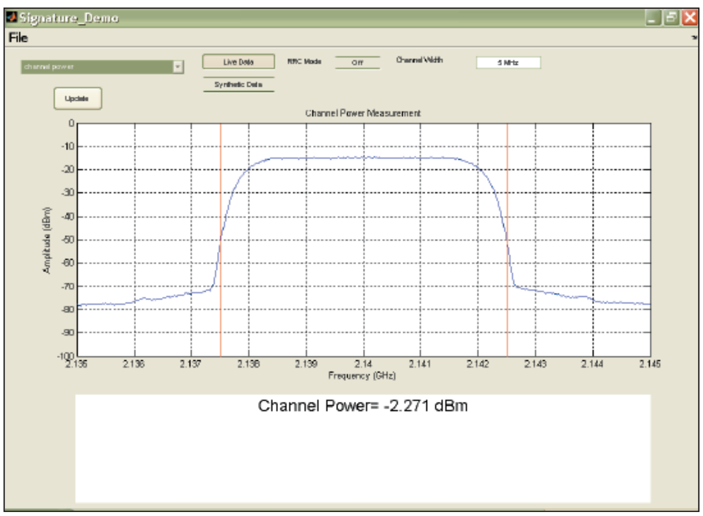
MATLAB Analysis Examples

These examples illustrate what is possible using Signature with MATLAB. The examples include making measurements of harmonics, channel power, adjacent-channel power ratio (ACPR), spectrograms, and custom demodulation. Results of these measurements are shown on this and the following page.

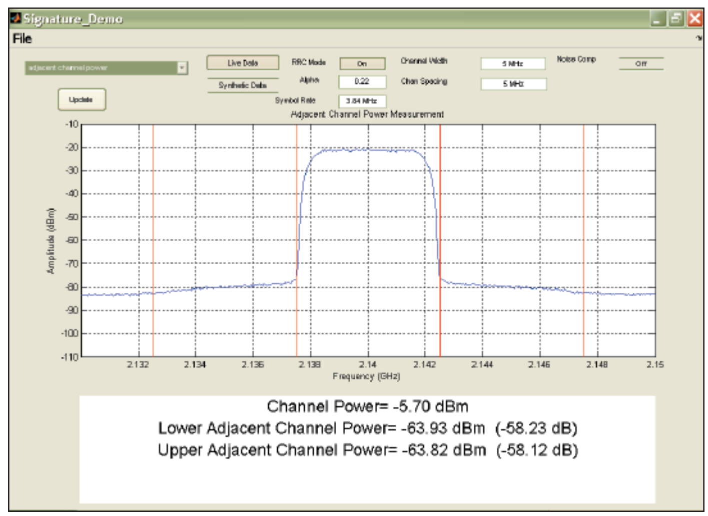
These displays are created using the MATLAB graphical-user-interface creation software, called GUIDE. This user interface is then combined with the measurement algorithms shown later in this application note. You can see all of these displays and more on Signature by using the demonstration code that comes with the “Connectivity to MATLAB” option (Option 40). For more details on this, refer to the section “MATLAB Demonstration.”



Example MATLAB harmonic measurement.



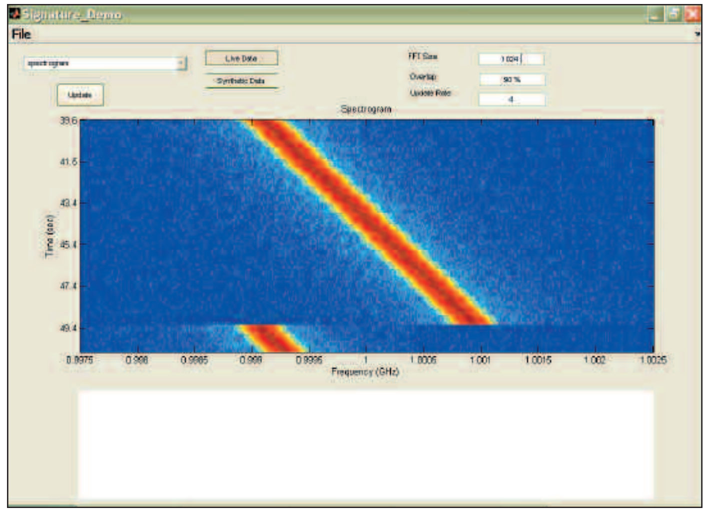
Channel Power measurement using MATLAB.



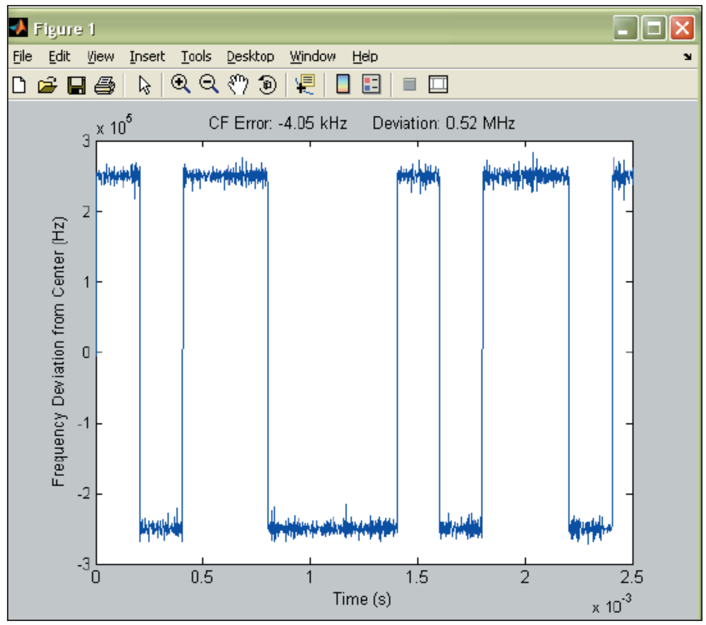
MATLAB Adjacent-Channel Power measurement.

Note that all of the example MATLAB code in this application note is available as part of the Signature "Connectivity to MATLAB" option, usually in the form of functions. You can tell the name of the function or script by referring to the 1st line of the example code.

All of this code can be found on Signature, in the directory: C:\Signature\MathworksConnectivity



MATLAB spectrogram of a swept signal.



FSK Modulation Measurements using MATLAB and Simulink.

Installing MATLAB on Signature

Install MATLAB with any options, using any MATLAB licensing option, onto the C: drive in Signature. It is best to install MATLAB into the default directory.

Note that for the seamless connectivity between Signature and MATLAB shown in this note, your Signature needs option 40—Connectivity to MATLAB.

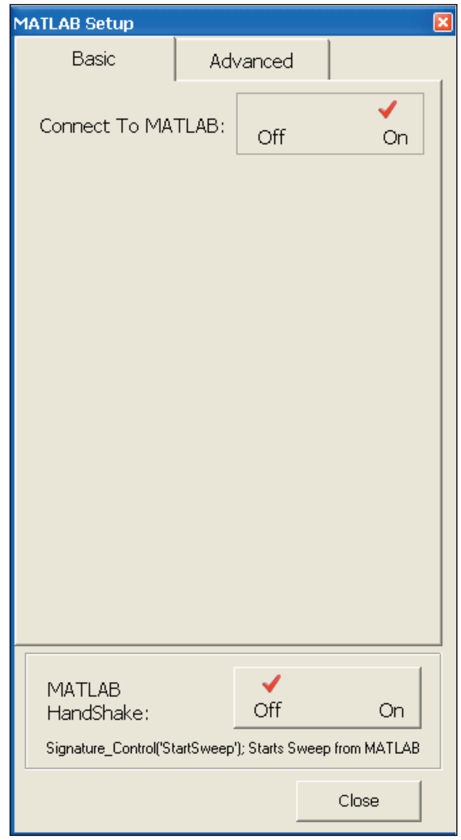
Configuring Signature to use MATLAB

Once you have MATLAB installed, start it from within Signature by clicking on the Signature Tools/MATLAB pulldown menu.

You will get all of the displayed Signature traces ported into MATLAB, for example as `Signature_Trace1` or `Signature_Trace2`. You can also get IQ vectors by using the Advanced tab in the dialog. See the 'IQ Vectors' section for more details on how to get IQ vectors.

You can then set up the instrument in the normal way. When sending traces to MATLAB, all active instrument traces and instrument setup data will all automatically appear in the MATLAB environment. This makes it much easier to get data out of the instrument into this industry-leading analysis tool.

You can also enable handshaking between Signature and MATLAB. Refer to page 10 for more details about this.

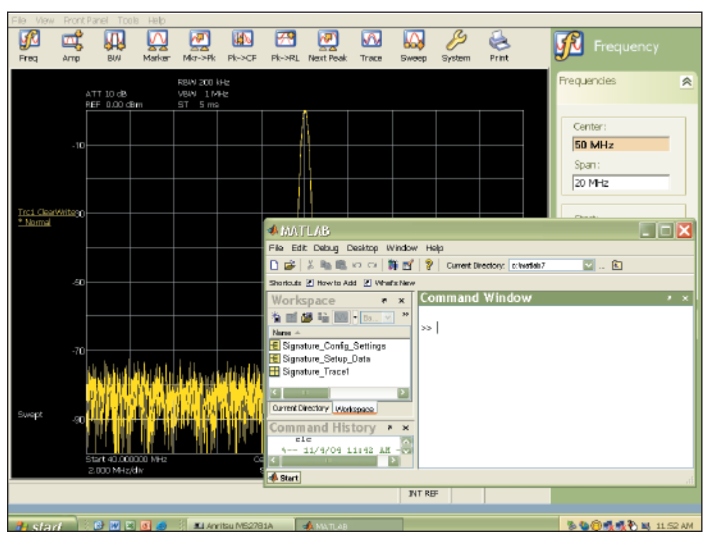


Signature Configuration dialog for connecting to MATLAB.

The MATLAB Desktop Window

When you start MATLAB from Signature, you get the normal MATLAB desktop. The Signature information is automatically available in the MATLAB Workspace, as you can see in the figure.

You can easily see the variables being used, type in commands, and see the history of commands you've used from this window.



MATLAB Desktop Window on Signature.

Getting Setup Information from Signature into MATLAB

When you start MATLAB from Signature, the instrument setup information is automatically created in a MATLAB structure called `Signature_Setup_Data`.

For example, if you need to know the center frequency that the instrument is tuned to, you can refer to this variable:

`Signature_Setup_Data.CenterFrequency`

This variable has the value (in Hz) of the current center frequency.

If you double click on a variable name in the Workspace pane on the MATLAB desktop, or type the variable name on a MATLAB command line, you can see the details of the structure and the current values.

Having the setup information automatically available in MATLAB means that you can set up the instrument to make measurements the normal way, and all of the setup information is conveniently available in MATLAB. This is much simpler than having to query the individual setup items, as you had to in the past.

```
>> Signature_Setup_Data

Signature_Setup_Data =

            CenterFrequency: 4.0000e+009
                       Span: 8.0000e+009
                  SweepTime: 16
                        RBW: 3000000
                        VBW: 10000000
            Reference_Level: 0
                Attenuation: 10
           Frequency_Offset: 0
     Reference_Level_Offset: 0
            ScaleTypeLinear: 0
                  SweepType: 'Normal'
                 dB_per_Div: 10
             Display_points: 501
                 Nbw_to_rbw: 1
                 SymbolRate: 3.8400
            SymbolRateUnits: 'MHz'
                InputSignal: 4
             ModulationType: 'QPSK'
            Sampling_period: 3.3000e-008
                  DataReady: 1
                  Handshake: 'Off'
```

Example Signature setup information in MATLAB.

Getting Data from Signature into MATLAB

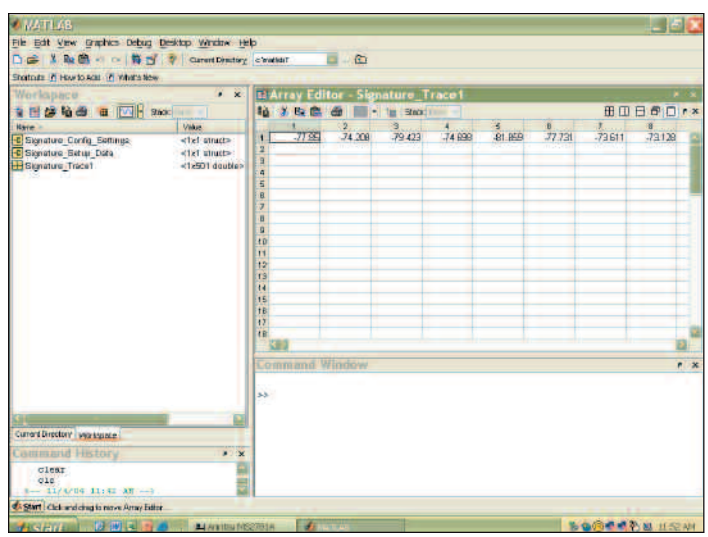
When you start MATLAB from Signature, you can make the active traces or IQ vectors automatically available in the MATLAB workspace. Then all you have to do is use them. In the next few pages there are a number of examples of how you might use the Signature data in MATLAB.

Viewing the trace values

If you double-click on a trace name (e.g. Signature_Trace1) in the MATLAB Workspace pane, you will see the values of the variable in the Array Editor pane.

These values are in the current measurement units, which you can check in the `Signature_Setup_Data` structure.

Note that the values in the Array Editor pane are only updated when you press Enter in the MATLAB Command Window, so instrument changes may not be immediately reflected there.



Double click on a variable in the Workspace pane to see the value(s) in the Array Editor pane.

Drawing a Signature trace in MATLAB

There are several ways to draw a Signature spectrum trace in the MATLAB environment, including using plot, using a loop, or using timers.

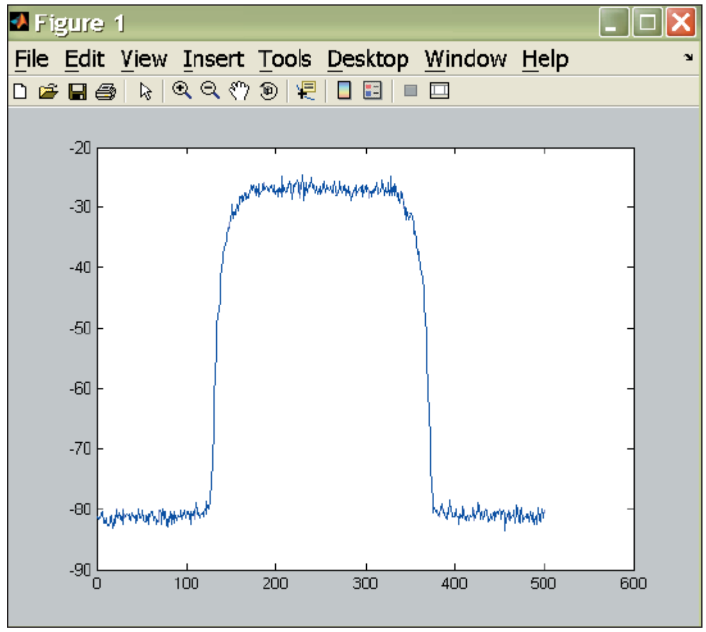
Plot

The simplest is to use the MATLAB plot function:

`plot(Signature_Trace1)`

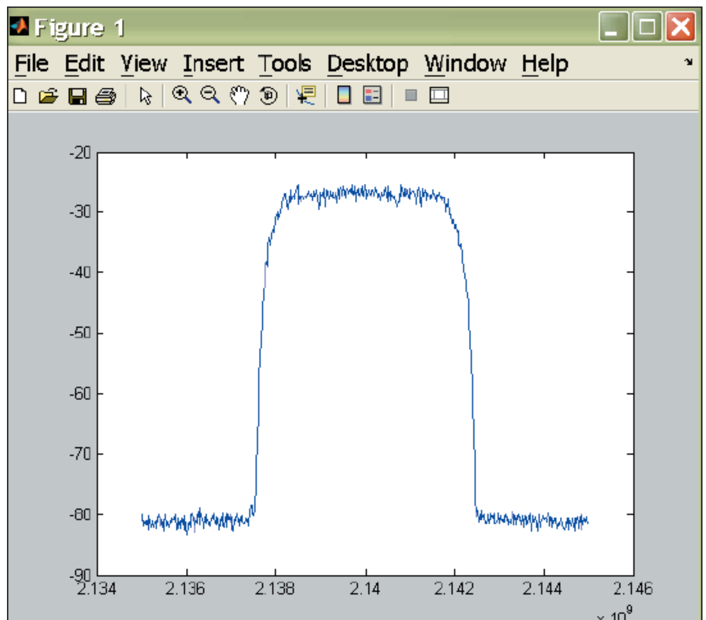
This is simple and effective, but doesn't scale the x-axis correctly, nor update the plot when new data is available.

Note that all active Signature traces are available, up to `Trace5`. Note that blank traces are not cleared from the Workspace.



Just type `plot (Signature_Trace1)` *to plot a trace.*

To properly scale the frequency axis, you need to create a vector that defines the frequency of each trace point. By using the information in the `Signature_Setup_Data` structure, this is simple—you just create a vector using the start and stop frequencies, and a step size based on the number of display points and the span . For example, the 7 lines of MATLAB code shown on the right create a function to do a scaled plot. The resulting plot is shown as well.




```
function PlotTrace_with_scaling(trace, setup)
%PlotTrace_with_scaling(trace,setup)
%Plot a Signature trace including x- & y-axis scaling
%trace is the Signature trace
%setup is the Signature setup structure

% Copyright 2004 Anritsu Company
% Revision 1.2  3 September 2004

start=setup.CenterFrequency - setup.Span/2;
stop=start + setup.Span;
display_intervals=setup.Display_points - 1;
freq_step=setup.Span / display_intervals;
freq=start : freq_step : stop;
plot(freq, trace);
```

A few lines of MATLAB code add a scaled x-axis.

Loops

By using a loop, you can automatically update the plot when the instrument updates the trace data. The figure at the right shows this. This code has two interesting aspects. The first is the loop statement, which is `while(ishandle(h))`. This loop repeats until the drawing window is closed, which is very natural for the user. The `pause(0.1)` statement allows MATLAB to actually draw the figure. MATLAB only updates graphics when computation isn't being done, so without the pause, the graph would never be updated. MATLAB would also consume all available CPU resources, slowing the instrument display.

```
% PlotTrace1
% Copyright 2004 Anritsu Company
% Revision 1.5  28 October 2004

plot(Signature_Trace1);
h=gca;
while (ishandle(h))
    PlotTrace_with_scaling_to_axis...
        (h, Signature_Trace1, Signature_Setup_Data)
    pause(0.1);
end
```

```
function PlotTrace_with_scaling_to_axis(h, trace, setup)
%PlotTrace_with_scaling(h, trace, setup)
%Plot a Signature trace including x- & y-axis scaling
%h is a handle to a pre-existing graphics axis
%trace is the Signature trace
%setup is the Signature setup structure

% Copyright 2004 Anritsu Company
% Revision 1.1  26 July 2004

start=setup.CenterFrequency - setup.Span/2;
stop=start + setup.Span;
display_intervals=setup.Display_points - 1;
freq_step=setup.Span / display_intervals;
freq=start : freq_step : stop;
plot(h, freq, trace);
```

A loop updates the graph automatically.

A slightly improved version of the above (shown at the right) doesn't re-draw the entire plot, but merely updates the trace data. This reduces the plotting overhead by about 50%.

By using the code `h=plot(Signature_Trace1)`, the variable h becomes a graphics handle that references the plot. The later code `set(h,'YData',Signature_Trace1)` updates the trace data, and the `pause(0.1)` makes MATLAB redraw the plot.

```
%PlotTrace1a

% Copyright 2004 Anritsu Company
% Revision 1.3 26 October 2004


start = Signature_Setup_Data.CenterFrequency - Signature_Setup_Data.Span/2;
stop = start + Signature_Setup_Data.Span;
display_intervals = Signature_Setup_Data.Display_points - 1;
freq_step = Signature_Setup_Data.Span / display_intervals;
freq = start : freq_step : stop;
plot(freq, Signature_Trace1);
h=get(gca,'Children');

while (ishandle(h))
    set(h,'YData',Signature_Trace1);
    pause(0.1);
end
```

Using `set` reduces plotting overhead by about 50%.

Timers

Loops let you have a live display, but only have one at a time and the MATLAB command line is blocked while this code is running. You can stop the code by hitting "Ctrl-C", but there is a more user-friendly way—by using a MATLAB timer. The MATLAB code below shows how to use a timer to re-plot the trace every 100 ms. With this code, if you close the figure window the timer will stop automatically.

To call this function, you must use the name of the variable, using either of the following two ways:

```
timerplot('Signature_Trace1')
timerplot Signature_Trace1
```

This is because MATLAB passes parameters to function by value. This means that once the function is called, you can't change the value of the parameter because the function has a copy of the data. To get around this, the timer code uses a MATLAB function called `evalin`. This function evaluates a MATLAB command and returns the current value. The timer code makes use of this by evaluating the current value of Signature_Trace1. An alternative to this would be to use a Global variable, but this is generally not good programming technique.

```
function timerplot(TraceName)
% function timerplot(TraceName)
% plots the data in the variable TraceName every 100 ms

% Copyright 2005 Anritsu Company
% Revision 1.2 27 April 2005

hFig = figure('Name',mfilename,'CloseRequestFcn',@closefig,'DoubleBuffer','on');
trace=evalin('base',TraceName);
hPlot = plot(trace);                        % Initial plot
title('timerplot');     xlabel('Frequency');    ylabel('Amplitude');

hTime = timer('Name',[mfilename,'Timer'],...    % Give it a name that coresponds to the file name
    'ExecutionMode','fixedSpacing',...          % Make the plot update on a fixed Rate
    'Period',0.1,...                            % Update the plot 10 times a second
    'StopFcn',@stoptimer,...                    % On Stop closes the figure and clear timer
    'TimerFcn',{@plotfunction,TraceName},...    % The real work of the plot
    'ErrorFcn',@ploterror,...                   % Error handling
    'UserData',[hFig,hPlot]);                   % Keep a handle to the plot to be updated.

builtin('set',hFig,'UserData',hTime);                    % Keep a handle to the timer
set(hFig,'HandleVisibility','callback');        % Hide the figure handle--other plots won't overwrite it
start(hTime);                                   % Start the timer

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function stoptimer(hTime,varargin)
udata = get(hTime,'UserData');
hFig=udata(1);
delete(hFig);
delete(hTime);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plotfunction(hTime,varargin)
udata = get(hTime,'UserData');                  % Retrieve handles to the figure & plot
hPlot = udata(2);
trace = evalin('base',varargin{2});             % Get the latest trace data
set(hPlot,'YData',trace);                       % Update the trace data (the y-axis)
drawnow;                                        % Force MATLAB to draw the plot

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function ploterror(varargin) % Get a handle to the figure and close it.
hFig = findobj('Name',mfilename);
if ~isempty(hFig)
    close(hFig);
end
disp(['error', varargin{:}]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function closefig(hFig,varargin) % Get a handle to the timer, stop & delete it, then close the figure
hTimer=get(hFig,'UserData');
try
    stop(hTimer);
catch
    closereq
end
```

A MATLAB timer lets you automatically update multiple plots as well as retain use of the MATLAB command line.

Synchronization

Another improvement of the above is to use the “Handshake” function in the Signature interface to MATLAB. This handshaking allows you to know when Signature is finished making a measurement. Handshaking can be useful for such things as storing or averaging multiple traces, where you need to know when the trace data is new. You can turn Handshake on or off from the checkbox on the bottom of the MATLAB setup dialog.

This figure shows how to plot traces with handshaking. The concept is simple—just wait for `Signature_Setup_Data.DataReady` to be set to 1. When you have copied or used the data from Signature, use the line `Signature_Control('StartSweep')`. This line of code calls a MATLAB function that has been added as part of the Signature Connectivity to MATLAB option, and it just tells Signature to start a new measurement. Some simple MATLAB code to use this functionality is:

```
while (Signature_Setup_Data.DataReady~=1 %New data ready?
      pause (0.01);                       %Give up CPU
end
%Code to use Trace or IQ Vectors
Signature_Control('StartSweep');          %Start a new sweep
```

In many of the examples, this is reordered somewhat for code simplicity, but this is the basic concept.

```
% PlotTrace1_sync
% Plots traces, syncronized with Signature handshaking
% Copyright 2005 Anritsu Company
% Revision 1.5  23 May 2005

plot(Signature_Trace1);
h=gca;                                    %Get a handle to the figure, so we know when it's closed
Signature_Control('StartSweep');          %Tell Signature to take a new measurement

while (ishandle(h))                       %Continue until figure closed
    PlotTrace_with_scaling_to_axis...
        (h, Signature_Trace1, Signature_Setup_Data)
    if strcmp(Signature_Setup_Data.Handshake,'On')  %Only look for Data Ready if Handshake on
        while Signature_Setup_Data.DataReady~=1     %Wait for Data Ready from Signature
            pause(0.01);                            %Allow Signature to use CPU
        end
        Signature_Control('StartSweep');            %Start a new measurement
    end
    pause(0.1);                           %Allow time for drawing
end
```

Plotting can also be synchronized with Signature sweeps.

Trace Averaging with Handshaking

We can then take the additions for synchronization and add an averaging function, seen in this figure. Note that due to the autoscaling in MATLAB, it may appear that averaging is not happening; check the y-axis scale to see that it reduces as the averaging progresses.

```
% PlotTrace1_avg
% Plots exponentially averaged traces
% Copyright 2005 Anritsu Company
% Revision 1.2  23 May 2005

Number_of_averages=10;
avg_factor=1/Number_of_averages;

plotted_trace=Signature_Trace1;
plot(plotted_trace);
h=gca;                          %Get a handle to the figure, so we know when it's closed

while (ishandle(h))
    PlotTrace_with_scaling_to_axis(h, plotted_trace, Signature_Setup_Data)
    plotted_trace=(1-avg_factor)*plotted_trace + ...
        avg_factor * Signature_Trace1;                %Perform exponential averaging
    Signature_Control('StartSweep');                  %Take a new measurement
    if strcmp(Signature_Setup_Data.Handshake,'On')   %Check that handshaking is turned on
        while Signature_Setup_Data.DataReady~=1      %Wait for Data Ready from Signature
            pause(0.01);
        end
    else
        disp ('Please turn Handshaking On in Signature for proper averaging');
        return
    end
    pause(0.1);
end
```

Synchronization between Signature & MATLAB allows trace averaging.

Storing Multiple Traces with Handshaking

Or we can use the synchronization to store multiple acquired traces, as shown in this figure. This lets you gather data quickly, then analyze later when you have more time. You could also store sets of captured IQ vectors in a similar fashion.

```
% StoreTraces
% Stores multiple traces to c:\Signature_Traces.mat
% Works best with Signature handshaking turned On

% Copyright 2005 Anritsu Company
% Revision 1.2  23 May 2005

Number_of_traces=10;
Trace_storage=zeros(Number_of_traces,length(Signature_Trace1)); %Pre-allocate storage array

for index=1:Number_of_traces
    if strcmp(Signature_Setup_Data.Handshake,'On')  %Check that handshaking is turned on
        while Signature_Setup_Data.DataReady~=1
            pause(0.01);
        end
        Trace_storage(index,:)=Signature_Trace1;    %Add current trace to storage array
        Signature_Control('StartSweep');
    else
        disp('Please turn on Handshaking to ensure stored traces are different');
    end
end
save('c:\Signature_Traces','Trace_storage','Signature_Setup_Data');
    %Store the captured traces & setup information to disk
```

Store multiple traces from MATLAB by using synchronization.

Manual Sweep with Handshaking

When you are synchronizing a MATLAB script to Signature, you may want to wait until the user pushes the Sweep key to do something new. For example, you may want to store traces from multiple experiments, where the user needs to change the device being tested.

In this case, you can replace the line that says

`Signature_Control('StartSweep');` with
`Signature_Setup_Data.DataReady=0;`

This keeps the code waiting until a new measurement is finished, which won't happen until after the user presses the Sweep key on Signature.

Timers with Handshaking

If we want to use both a timer and the Signature handshaking function, we need to modify the timer code a bit. The changes are very simple, so we haven't reproduced the entire code here. Only the 'plotfunction' is modified by adding the code to check if the Handshake is on, wait for DataReady, and start a new sweep. The function is also called with both the trace name and setup name, such as:

```
timerplot_sync('Signature_Trace1','Signature_Setup_Data')
```

If you wish, you can look at the complete code on a Signature with option 40.

```
function plotfunction(hTime,varargin)
udata = get(hTime,'UserData');                    % Retrieve handles to the figure & plot
hPlot = udata(2);

setup_structure = evalin('base',varargin{3});
if strcmp(setup_structure.Handshake,'On') && setup_structure.DataReady~=1
    return    % If Handshake is on and data isn't ready, don't plot
end

trace = evalin('base',varargin{2});           % Get the latest trace data
set(hPlot,'YData',trace);                     % Update the trace data (the y-axis)
drawnow;                                      % Force MATLAB to draw the plot
if strcmp(setup_structure.Handshake,'On')   %Only start a new sweep if there is handshaking
    Signature_Control('StartSweep');
end
```

By changing the `plotfunction` *in* `timerplot.m`*, timer-based plots can work with synchronization.*

Zero-span traces

For a zero-span trace, the Y-axis is identical to a spectrum trace, but the span is zero, and the X-axis is now in time. You can again use the `Signature_Setup_Data` structure to check the span, and then find the trigger delay and time-per-division values.

The code below shows an expanded version of the previous PlotTrace code that also plots scaled zero-span traces.

```
function PlotTrace_with_scaling_and_zero_span(trace,setup)
%Plot a Signature trace including x- & y-axis scaling
%for both spectrum & zero-span traces

% Copyright 2004 Anritsu Company
% Revision 1.2 25 August 2005


if setup.Span == 0 % zero span, so x-axis is time
    start_time = 0;
    stop_time = start_time + setup.SweepTime;
    display_intervals = setup.Display_points - 1;
    time_step = setup.SweepTime / display_intervals;
    time = start_time : time_step : stop_time;
    plot(time, trace);

else
    start=setup.CenterFrequency - setup.Span/2;
    stop=start + setup.Span;
    display_intervals=setup.Display_points - 1;
    freq_step=setup.Span / display_intervals;
    freq=start : freq_step : stop;
    plot(freq, trace);
end
```

Example MATLAB code to plot either spectrum or zero-span traces.

Modulation Measurements

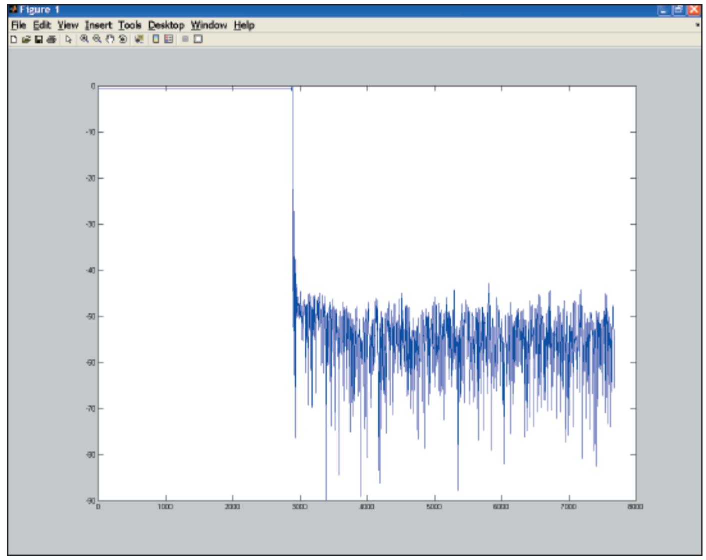
Traces from the modulation measurements option (option 38) are created as separate variable names.

These are:

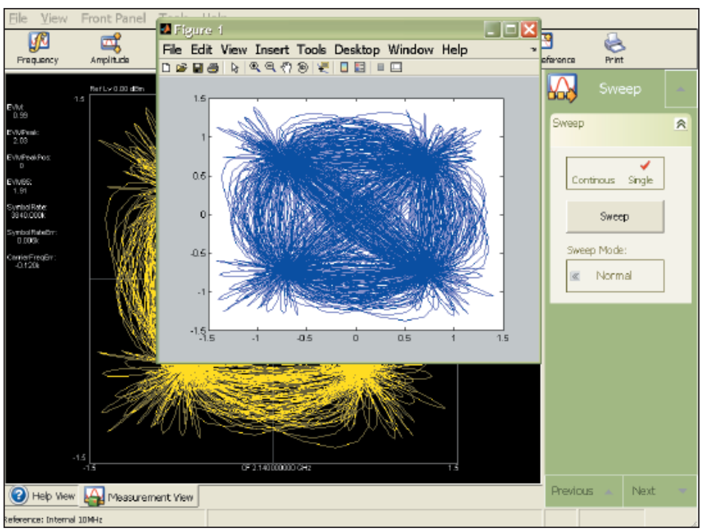
`Signature_VsaTimeDomain_Data`
`Signature_VsaVector_Data`

The `Signature_VsaTimeDomain_Data` is the power versus time waveform. This is normalized so that the peak value is 0 dB. You can see a plot of this in the figure to the right. This graph was created by using `plot(Signature_VsaTimeDomain_Data)`

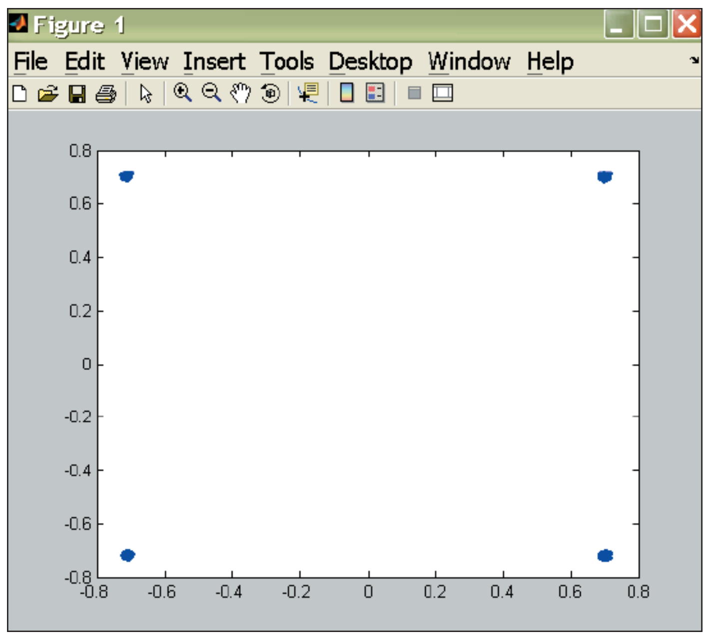
The `Signature_VsaVector_Data` is the vector diagram waveform. You can plot a constellation by selecting the symbol points and plotting just 'markers' in MATLAB, as shown in the code below.



A plot of `Signature_VsaTimeDomain_Data` shows the equivalent of a zero-span waveform.



MATLAB & Signature Vector Diagrams.



MATLAB Constellation plot.

```
%Constellation_plot
% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

points_per_symbol=8;
constellation_points = reshape(Signature_VsaVector_Data,points_per_symbol,[]);
constellation_points = constellation_points(1,:);
plot(constellation_points,'LineStyle','none','Marker','.')
```

Plot a constellation in MATLAB with this code.

IQ Vectors

The IQ vectors from Signature give you the most freedom to make complex measurements, such as FFTs or demodulation. The IQ vectors also allow larger data sets, such as for making CCDF measurements (refer to the CCDF section for more details on this measurement). You can get up to 10 million IQ vectors in a few seconds, as well as 30 MHz capture bandwidth if the Signature has Option 22. If you enable IQ vector output, you will get a MATLAB variable called `Signature_IQ_Data`, as well as the setup structure `Signature_Setup_Data`.

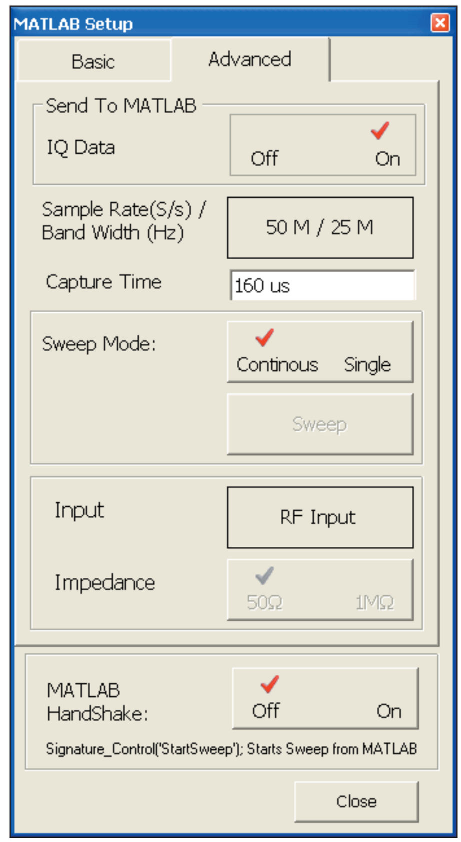
Having easy access to IQ vectors provides the ultimate capability in creating custom measurements. Since the IQ vectors describe the I and Q state (or equivalently the amplitude and phase), you can extract any information about the signal that you want. You can determine the frequency variation versus time, create spectrograms, look at amplitude statistics, or even demodulate the signal. Measurements that describe how to do this are shown later in this note.

To send IQ vectors to MATLAB, select the 'Advanced' tab on the MATLAB setup dialog in Signature. Then turn on Send to MATLAB IQ Data. You can now select the sample rate, capture length, choose single or continuous sweep, and pick the input source (RF or rear-panel IQ). Handshaking is still available in the IQ vector output mode.

You may want to note several things about the IQ vector output to MATLAB mode:

- The sample rate that you choose is the sample rate of the IQ vectors output to MATLAB. Note that this is 1/2 of the sample rate of the IF signal inside Signature, before the signal is converted to IQ vectors.
- No traces are displayed; the instrument is now dedicated to output IQ vectors to MATLAB.
- If you close the Signature MATLAB dialog, the instrument automatically exits IQ vector output mode; this ensures you don't get a blank display.
- There are no calibrations applied to the IQ vectors. This means that the absolute amplitude may be off by several dB, and that the frequency response (especially near band edges) may vary in both amplitude and phase. The frequency response over the center 10% of each band is very flat, however.
- For the fastest sample rates there may be transients at the beginning and end of the data. For the 12.5 and 25 MS/s sample rates, there is a transient at both the beginning and end that is about 20 samples long. For the 50 MS/s rate, the beginning transient is very small, and is only about 5 samples long; at the end of the data the transient is about 10 samples long. If you are using these sample rates, you may want to eliminate these points from you measurements. The use of negative trigger delay (pre-trigger) and extending the capture time can help with this.

  There are no visible transients for lower sample rates.



Signature MATLAB Setup dialog for IQ vector output.

Plotting IQ Vectors

Some simple examples of using IQ vectors are plotting the amplitude of the signal versus time, the spectrum of the signal (by using an FFT), and the amplitude of the individual I and Q waveforms.

Plotting the Magnitude of IQ vectors

The figure below shows how to plot the amplitude of the IQ signal; this is the same as the envelope of the signal into Signature. This allows handshaking, but instead of plotting the traces, it plots `10*log10(abs(Signature_IQ_Data))`

```
% Plot_IQ
% Copyright 2005 Anritsu Company
% Revision 1.2  23 May 2005

plot(abs(Signature_IQ_Data));
h=gca;
Signature_Control('StartSweep');                    %Tell Signature to take a new measurement

while (ishandle(h))
    plot(10*log10(abs(Signature_IQ_Data)));
    if strcmp(Signature_Setup_Data.Handshake,'On') %Only start a new sweep if there is handshaking
        while Signature_Setup_Data.DataReady~=1
            pause(0.01);
        end
        Signature_Control('StartSweep');
    end
    pause(0.1);
end
```

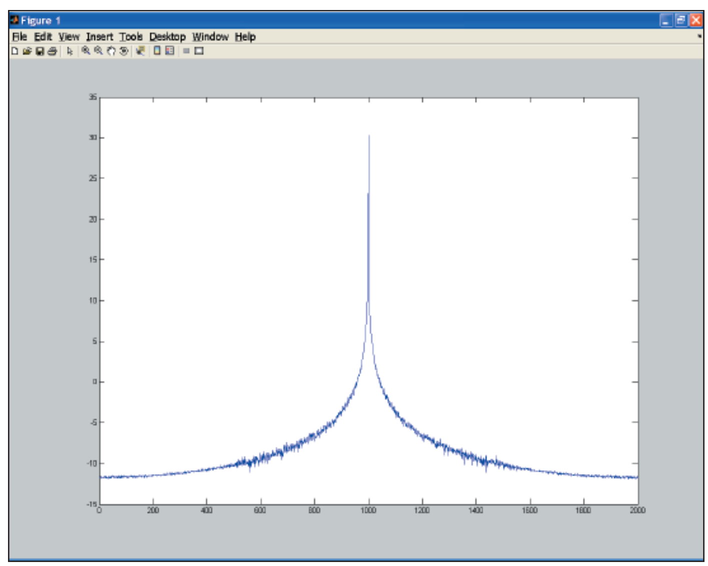
Plot the magnitude of IQ vectors.

FFT of IQ vectors

You can also plot the spectrum of the IQ vectors by changing the plot command to:

`plot(10*log10(abs(fftshift(fft(Signature_IQ_Data)))));.`

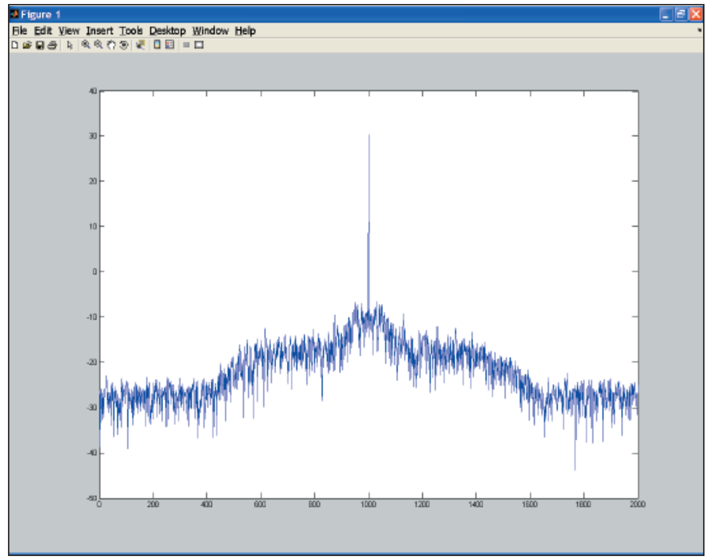
An example of the results is shown in the figure to the right.



FFT of IQ vectors, without windowing.

FFT with Windowing

An enhancement to this is to use an FFT window, such as the well-known 'Hann' window. Many windows are available in the MATLAB Signal Processing Toolbox. These windows reduce the "leakage" or the sidebands on the signal above. Different windows have different effects on the spectrum. A complete discussion of window choices is beyond the scope of this technical note. An example of a windowed FFT is shown in the figure at the right. Note that the sidelobes are about 20 dB lower than in the previous FFT plot.



FFT of IQ vectors, using a Hann window.

To use a window, replace the FFT "`plot…`" line on the previous page with the following 5 lines. An automatic FFT plot of the windowed IQ vectors is available by calling the script `plot_IQ_fft`.

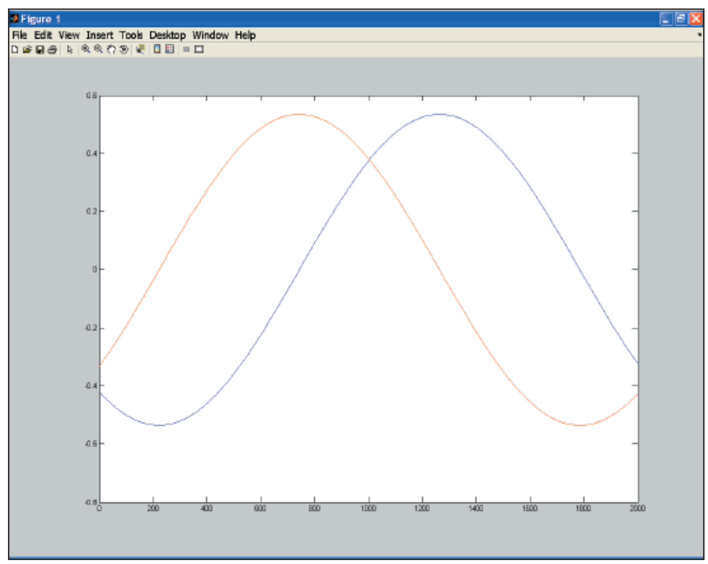
```
plot_IQ_fft.WindowLength=length(Signature_IQ_Data);
WindowArray=window(@hann,length(Signature_IQ_Data))';
WindowAmplitudeCorrection=WindowLength/sum(WindowArray);
Trace=10*log10(abs(fftshift(fft(Signature_IQ_Data.*…
      WindowArray*WindowAmplitudeCorrection))));
plot(Trace);
```

I and Q magnitudes

Sometimes you want to see the I and Q waveforms directly. You can do this by replacing the plotting lines with:

```
plot(real(Signature_IQ_Data));
hold on;
plot(imag(Signature_IQ_Data),'-r');
hold off;
```

This is available by calling the script `Plot_I_and_Q`.



Overlaid plots of the I & Q waveforms.

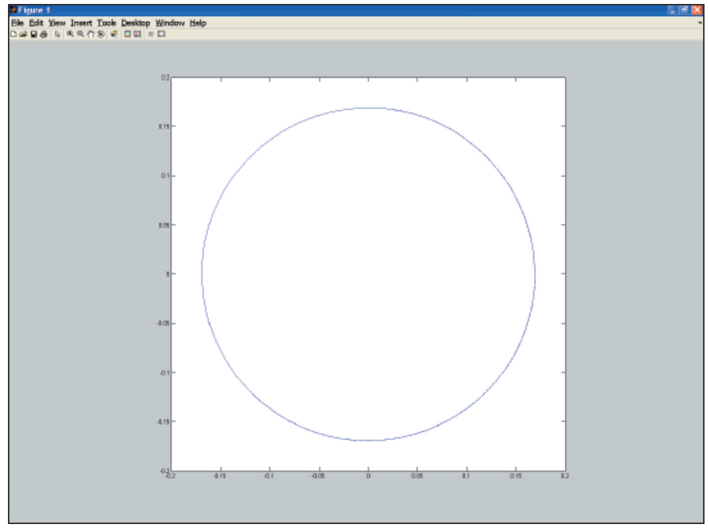
I and Q Polar plot

You may also want a polar plot of the IQ vectors. Since MATLAB automatically makes polar plots of complex variables, you can do this by replacing the plot lines with

```
plot(Signature_IQ_Data);
set(h,'DataAspectRatio',[1 1 1]);
```

The second line is necessary to make the I and Q axes equal.

This is available by calling the script `Plot_IQ_polar`.



Polar plot of IQ vectors.

Saving captured IQ vectors to the Anritsu MG3700A Vector Signal Generator

A powerful use of captured IQ vectors is to replay them. You can do this with an arbitrary-waveform based Vector Signal Generator, such as the Anritsu MG3700A. The code to the right, `csvout`, saves the Signature IQ vectors in an ASCII file. This file is suitable for loading into the MG3700A by using the 'Convert' function in the IQProducer software that comes with the MG3700A.

```
% csvout writes a CSV file of Signature IQ vectors for
% use with the Anritsu MG3700A Vector Signal Generator
% The output file is 'C:\signature_IQ'

% Version 1.0
% Copyright 12 May 2005
% Anritsu Company

csvwrite('c:\signature_IQ',...
    [real(Signature_IQ_Data)',...
    imag(Signature_IQ_Data)',...
    [ones(1,ceil(length(Signature_IQ_Data)/2)),...
    zeros(1,floor(length(Signature_IQ_Data)/2))]'
    ]);
```

Write a CSV file to prepare captured IQ vectors & triggering for the MG3700A Vector Signal Generator.

Since the captured signal is probably not periodic, there will be a "glitch" at the end of the waveform as it wraps around to the beginning. There are several ways to deal with this:

- Capture as long a waveform as possible. With Signature, you can capture up to 10 million IQ vectors, and the MG3700A has an even longer memory available. By capturing a longer waveform, the glitch doesn't happen as often, and therefore has lower power.
- Use a trigger signal for measurements. The MATLAB code shown here creates a signal out of the rear panel of the MG3700A connector labeled "Connector 1". This signal goes high at the beginning of the waveform and has about 50% duty cycle. You can use this signal to trigger Signature or other measurement equipment, so that you can avoid the wrap-around glitch.
- Capture a bursted signal with triggering. If the signal is bursted, you can make the beginning and end of the waveform almost identical by ensuring the waveform is off at these points—leaving only noise. This makes the wraparound glitch energy very small. You can use the triggering functions on Signature, including pre-trigger (negative trigger delay), to capture just the bursted part of the signal.
- Capture a periodic signal. If you can capture exactly N periods of the signal, there won't be a wraparound glitch. There are 2 ways to do this:
  - If you can phase lock Signature and the source to the same reference frequency, and if there is an integer relationship between a Signature sample rate and the Device Under Test symbol rate, you can acquire a set of samples that describe exactly "N" periods of the signal. For example, if the DUT symbol rate is 101 kHz, and you pick the 1 MHz sample rate in Signature, every 1000 samples in Signature will be exactly 101 symbols from the DUT.
  - If the DUT has a modulation format that Signature can demodulate (using the Modulation Analysis option, option 38), then there is no need to phase lock or pick sample rates carefully. Just demodulate the signal and use the `Signature_VsaVector_Data` output. To write this file, edit the `csvout` code to use `Signature_VsaVector_Data` instead of `Signature_IQ_Data`.
- "Window" the signal. This is similar in concept to the Windowing used for FFTs. By tapering the ends of the waveform to zero, you can reduce the wraparound glitch energy. This does, however, add low-rate amplitude modulation to the signal, which may or may not be acceptable for your use. By capturing a longer signal, you can reduce the rate of this amplitude modulation.

Example Applications

Spectral Measurements

Signature has a number of built-in spectral measurements, such as channel power and adjacent channel power ratio (ACPR). This section shows you how to do these and more in MATLAB. This gives you the ultimate flexibility in creating your own custom measurements. For illustration purposes, this section shows measurements of:

- Channel power in several ways:
  - Without channel filtering
  - With channel filtering
  - As a script
  - As a function
- Adjacent channel power
- Improved channel power and adjacent channel power measurements using a concept called noise compensation.
- Multi-carrier power
- Harmonics
- Occupied Bandwidth
- Power Spectral Density

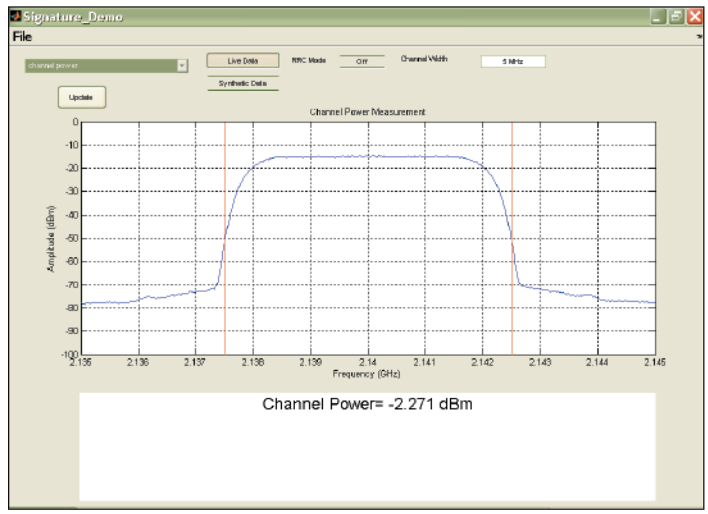
Channel Power

The Channel Power measurement is used for measuring the power of a digitally modulated signal. It is simply an integration of the trace across the channel width, plus corrections for the resolution bandwidth. The instrument should be set up for RMS detection (via the Trace menu) to make an accurate measurement.

The MATLAB code below computes:

- Which trace points encompass the channel
- A correction based on the noise bandwidth of the current RBW filter
- The trace in mW (instead of dBm)
- The uncorrected channel power, in mW
- The corrected channel power, in dBm

The computed channel power in dBm is then displayed.



Channel Power measurement using MATLAB.

```
%channel_power -- measures power over a defined bandwidth

% Copyright 2005 Anritsu Company
% Revision 1.2 13 May 2005

cbw=5e6;                                                % Set the channel bandwidth

trace=Signature_Trace1;                                 % Extract Signature data for readability
span=Signature_Setup_Data.Span;
rbw=Signature_Setup_Data.RBW;
nbw_cor=Signature_Setup_Data.Nbw_to_rbw;


channel_points = length(trace) * cbw/span;              % Find the # of trace points that describe a channel
channel_center = ceil(length(trace)/2);                 % Find the trace center point, =251 for 501 points
channel_start = channel_center - floor(channel_points/2);  % Find the channel start point
channel_stop  = channel_center + floor(channel_points/2);  % Find the channel stop point

power_cor = ( cbw / (rbw/nbw_cor) ) * (1/channel_points);  % Compute correction factor for noise-like signals

power_trace = 10.^(trace/10);                           % Convert trace to mWatts
integ_pwr = sum( power_trace(channel_start:channel_stop) ); % Integrate the power over the channel
power = 10*log10( power_cor * integ_pwr );              % Correct power value for noise-like signals
```

Channel power is a simple computation in MATLAB.

Channel Power with Filtering

Some channel power measurements require using a receiver filter, such as for the UMTS system. If we modify the above code, we can easily add this filtering function. Once the filter is created (in the frequency domain), you must multiply the spectrum (in mW) by the filtering function using this line of code:

```
power_trace = power_trace .* rrc_filter;
```

An example of creating a root-raised cosine (RRC) filter, such as used in the UMTS system is shown below:

```
function y = rrc(tot_span, sym_rate, alpha, display_points)
% y = rrc(tot_span, sym_rate, alpha, display_points)
% Calculates and returns the freq response 'y' for a Root Raised Cosine
% filter with parameter alpha. The freq response is evaluated
% over a span that equals tot_span and the 2 sided bandwidth for the filter
% is sym_rate
% The response for an ideal RRC filter is 1 in the passband, 0 in the stop
% band and 0.7071 at frequencies = half the sym_rate

% Copyright 2004 Anritsu Company
% Revision 1.1 27 April 2005
% Modified from revision 1.0 so that the RRC function applies to power waveforms,
% rather than voltage waveforms.

wc = 2*pi*(sym_rate/2);
span = tot_span/2;

f = -span:(2*span/(display_points-1)):span;
w = 2*pi*f;

y=zeros(1,length(w));   %pre-allocate the array for speed
for i = 1: length(w)
    if (abs(w(i)) < wc*(1-alpha))
        y(i) = 1;
    elseif (abs(w(i)) > wc*(1+alpha))
        y(i) = 0;
    else
        y(i) = ( (1/2) + (1/2)*cos( pi*(abs(w(i))-wc*(1-alpha)) / (2*alpha*wc) ) );
    end
end
```

MATLAB root-raised cosine (RRC) filter creation function (in the frequency domain).

Channel Power *Function* with Optional Filtering

If we combine all of the above pieces, add noise bandwidth correction, and then create MATLAB functions for channel power and the RRC filter we get the following:

```
function power = cp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate)
% channel power compuatation for noise-like signals
% y = cp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate)
% Calculates the channel power (in dBm) for the given trace data and cbw (channel bandwidth)
% trace is assumed to be a vector of dBm values.  Span and rbw are the
% associated span and rbw for the trace.
% nbw_cor is the noise bandwidth correction value, in linear terms
%
% rrc_mode can be 'on' or 'off', alpha is the roll-off factor for the RRC
% filter and sym_rate is the symbol rate that is used in the filter

% Copyright 2004 Anritsu Company
% Revision 1.2  11 May 2005

channel_points = length(trace) * cbw/span;                 % Find the # of trace points that describe a channel
channel_center = ceil(length(trace)/2);                    % Find the trace center point, =251 for 501 points
channel_start = channel_center - floor(channel_points/2);  % Find the channel start point
channel_stop  = channel_center + floor(channel_points/2);  % Find the channel stop point
power_cor = ( cbw / (rbw/nbw_cor) ) * (1/channel_points);  % Compute correction factor for noise-like signals

power_trace = 10.^(trace/10);                              % Convert trace to mWatts

if (strcmp(rrc_mode,'on'))
    rrc_filter = rrc(span, sym_rate, alpha, length(trace)); % Create RRC channel filter
    power_trace = power_trace .* rrc_filter;               % Apply channel filter (if requested)
end

integ_pwr = sum( power_trace( channel_start : channel_stop ) );%Integrate the power over the channel
power = 10*log10( power_cor * integ_pwr );                 % Correct power value for noise-like signals
```

MATLAB channel power function, with optional RRC filtering.

To invoke the channel power function for a UMTS signal with the required RRC filtering and over the defined 5 MHz channel width, then display the result (in dBm), type the following (all on one line):

```
cp(5e6, Signature_Setup_Data.RBW,...
Signature_Setup_Data.Span, Signature_Trace1, ...
Signature_Setup_Data.NBW_to_rbw, 'on', 0.22, 3.84e6)
```

Then MATLAB will respond with an answer, such as:

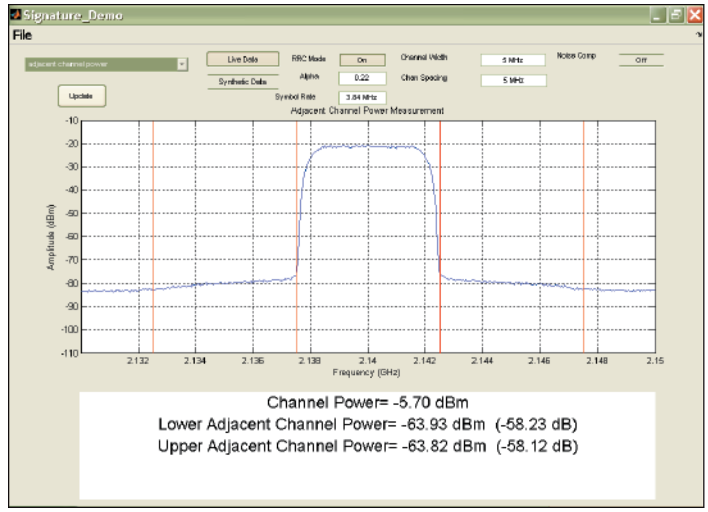
```
 ans=
-10.00
```

Adjacent Channel Power (ACP)

The concept of channel power is easy to extend to adjacent channel power. We can extend the above channel power function by calling it several times—once for the channel and once for each adjacent channel. The below MATLAB function does exactly this.

This code simply calls the channel power function (defined in the Channel Power section above), then takes a portion of the spectrum trace for computing the power of the adjacent channels. The result is the power in each channel. To convert this to the Adjacent Channel Power Ratio, simply subtract the ACP levels from the channel power.

Signature's 27 dBm typical Third-Order Intercept (TOI) and low noise figure allow accurate ACPR measurements of high performance devices.



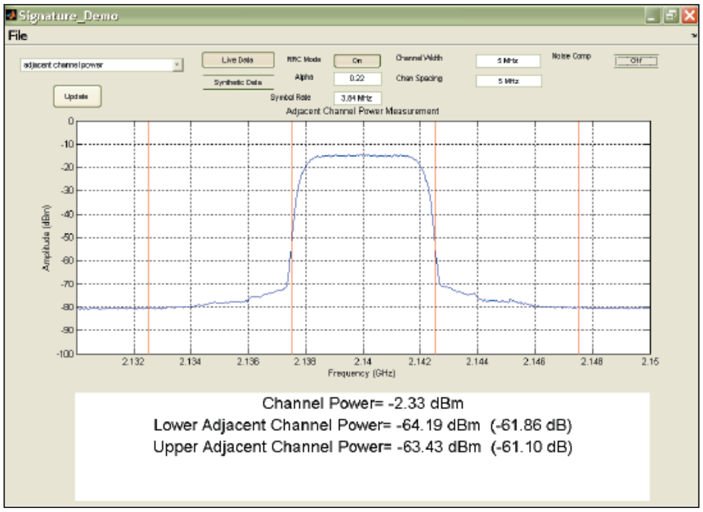
MATLAB Adjacent-Channel Power measurement.

```
function [ch_pow, acp_l, acp_r] = acp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate, channel_spacing)
% acp=adjacent channel power, used for acpr & aclr
% [ch_pow acp_l acp_r] = acp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate, channel_spacing)
% ch_pow is the channel power (in dBm) for the given trace data and cbw (channel bandwidth)
% acp_l is the channel power (in dBm) for the left adjacent channel
% acp_r is the channel power (in dBm) for the right adjacent channel
%
% trace is assumed to be a 1-D vector of dBm values and span,rbw is the
% associate span and rbw for the trace
% nbw_cor is the noise bandwidth correction value, in linear terms
%
% rrc_mode can be 'on' or 'off', alpha is the roll-off factor for the RRC
% filter and sym_rate is the symbol rate that is used in the filter
% The RRC filter is applied to the primary channel

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

ch_pow = cp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate);

freq_per_point = span/length(trace);

center = ceil(length(trace)/2);
l_center = round(center - (channel_spacing/freq_per_point));
r_center = round(center + (channel_spacing/freq_per_point));

l_trace = trace( (l_center - floor((cbw/2)/freq_per_point)) : (l_center + floor((cbw/2)/freq_per_point)) );
r_trace = trace( (r_center - floor((cbw/2)/freq_per_point)) : (r_center + floor((cbw/2)/freq_per_point)) );

acp_l = cp(cbw, rbw, cbw, l_trace , nbw_cor, rrc_mode, alpha, sym_rate);
acp_r = cp(cbw, rbw, cbw, r_trace , nbw_cor, rrc_mode, alpha, sym_rate);
```

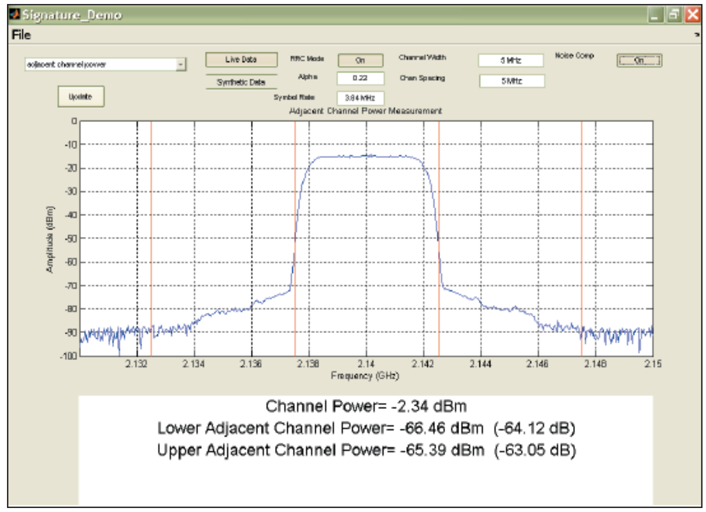
MATLAB Adjacent Channel Power function.

Noise Compensation

When measuring ACP, often the instrument noise floor is a limiting factor in making the most accurate measurements. This limitation can be reduced by using "noise compensation". This simply measures the instrument noise floor, using RMS detection and long sweep time to reduce the variance due to noise. The function on the next page modifies the above ACP measurement by subtracting a reference trace of just the noise from the measured trace including the signal.



MATLAB Adjacent Channel Power measurement without noise compensation.



MATLAB Adjacent Channel Power measurement with noise compensation. Notice the lower sideband levels.

```
function [ch_pow, acp_l, acp_r, trace] = acpnc(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate, channel_spacing, noise_trace)
% acpnc=adjacent channel power, used for acpr & aclr; noise compensation is
% added to be able to measure devices with better specifications
% [ch_pow acp_l acp_r, output_trace] = acpnc(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate, channel_spacing, noise_trace)
% ch_pow is the channel power (in dBm) for the given trace data and cbw (channel bandwidth)
% acp_l is the channel power (in dBm) for the left adjacent channel
% acp_r is the channel power (in dBm) for the right adjacent channel
% trace (as an output) is the result of subtracting the noise trace from the input trace
%
% trace is assumed to be a 1-D vector of dBm values and span,rbw is the
% associate span and rbw for the trace
% nbw_cor is the noise bandwidth correction value, in linear terms
%
% rrc_mode can be 'on' or 'off', alpha is the roll-off factor for the RRC
% filter and sym_rate is the symbol rate that is used in the filter
% The RRC filter is applied to the primary channel
%
% Noise compensation is used to reduce the instrument noise floor.  Measure
% the noise floor with the signal disconnected, and supply this trace as
% noise_trace

% Copyright 2004 Anritsu Company
% Revision 1.2 10 August 2005

noise_trace=10.^(noise_trace./10);
trace=10.^(trace./10)-noise_trace;
noise_trace=0.01.*noise_trace; %create clip limits 20 dB lower than the noise trace
trace=10*log10(max(trace,noise_trace));

ch_pow = cp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate);

freq_per_point = span/length(trace);

center = ceil(length(trace)/2);
l_center = round(center - (channel_spacing/freq_per_point));
r_center = round(center + (channel_spacing/freq_per_point));

l_trace = trace( (l_center - floor((cbw/2)/freq_per_point)) : (l_center + floor((cbw/2)/freq_per_point)) );
r_trace = trace( (r_center - floor((cbw/2)/freq_per_point)) : (r_center + floor((cbw/2)/freq_per_point)) );

acp_l = cp(cbw, rbw, cbw, l_trace , nbw_cor, rrc_mode, alpha, sym_rate);
acp_r = cp(cbw, rbw, cbw, r_trace , nbw_cor, rrc_mode, alpha, sym_rate);
```

MATLAB function for measuring ACP with noise compensation.

Plotting a Trace and Measuring ACP with Noise Compensation

By combining several of the above concepts, we can get a graph of the instrument trace including labels showing the ACP. The following code shows:

- Capturing a reference trace
- Calling the `acpnc` function to compute ACP using noise compensation
- Plotting the trace after the noise is removed
- Labeling the plot with the channel power ratios
- Using the set function to speed the plot updates

```
% acpr_nc  adjacent channel power ratio (acpr) compuation & graphing with noise compensation

% Copyright 2004 Anritsu Company
% Revision 1.1  12 October 2004

cbw = 5e6;
rrc_mode = 'on';
alpha = 0.22;
sym_rate = 3.84e6;
channel_spacing = 5e6;

rbw = Signature_Setup_Data.RBW;
span = Signature_Setup_Data.Span;
nbw_cor = Signature_Setup_Data.Nbw_to_rbw;
start = Signature_Setup_Data.CenterFrequency - Signature_Setup_Data.Span/2;
stop = start + Signature_Setup_Data.Span;
display_intervals = Signature_Setup_Data.Display_points - 1;
freq_step = Signature_Setup_Data.Span / display_intervals;
freq = start : freq_step : stop;


input ('Disconnect input signal, let the sweep finish, then press enter');
noise_trace=Signature_Trace1;
input ('Reconnect input signal, then press enter');
trace=Signature_Trace1;

[ch_pow, acp_l, acp_r, trace] = acpnc(cbw, rbw, span,  trace, nbw_cor, rrc_mode, alpha,...
                                      sym_rate, channel_spacing, noise_trace);
h=plot(freq, trace);
title('Spectrum Plot with Noise Compensation');
xlabel ([ 'acp lower:' num2str(acp_l-ch_pow) ' dB       '...
              'channel power: ' num2str(ch_pow) ' dBm       '...
              'acp higher: ' num2str(acp_r-ch_pow) ' dB']);

while(ishandle(h))
   trace=Signature_Trace1;
   [ch_pow, acp_l, acp_r, trace] = acpnc(cbw, rbw, span,  trace, nbw_cor, rrc_mode, alpha,...
                                         sym_rate, channel_spacing, noise_trace);
   set(h,'YData',trace)
   xlabel ([ 'acp lower:' num2str(acp_l-ch_pow) ' dB       '...
              'channel power: ' num2str(ch_pow) ' dBm       '...
              'acp higher: ' num2str(acp_r-ch_pow) ' dB']);
   pause(0.1);
end
```

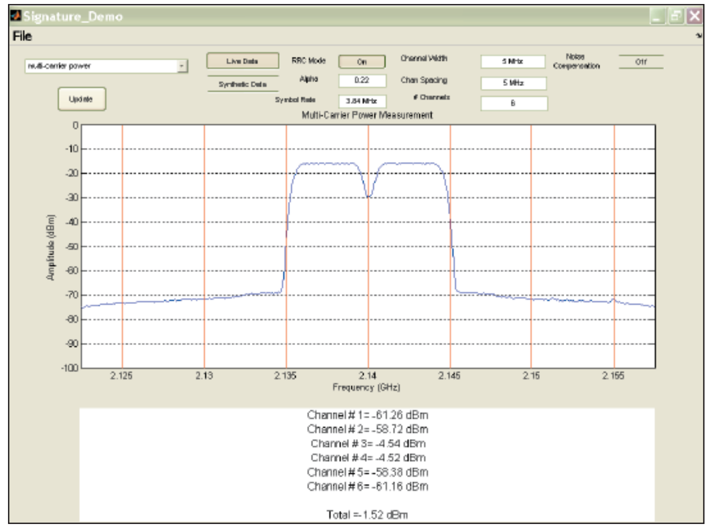
Plot a trace & measure ACP with noise compensation with this MATLAB code.

Multi-Carrier Power

The ACP function can easily be expanded to show the power in multiple carriers and offsets. The below code does this by calling the `cp` function multiple times, once for each channel, and then reporting all of the power levels as a vector.

The second set of code below adds the noise compensation function, just like for ACP.

Again, Signature's exceptional TOI performance and low noise floor enable accurate, fast multi-carrier power measurements.



Multi-Carrier Power measurement.

```
function ch_powers = mcp(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate, channel_spacing, num_channels)
% multi-carrier power measurement
% ch_powers = mcp(cbw, rbw, span, trace, rrc_mode, alpha, sym_rate, channel_spacing)
% ch_powers is a vector of the channel power (in dBm) for each channel in the given trace data
%
% trace is assumed to be a 1-D vector of dBm values and span,rbw is the
% associated span and rbw for the trace
% nbw_cor is the noise bandwidth correction value, in linear terms
%
% rrc_mode can be 'on' or 'off', alpha is the roll-off factor for the RRC
% filter and sym_rate is the symbol rate that is used in the filter
% The RRC filter is applied to each channel
% channel_spacing is the width of each channel
% The spectrum is assumed to be centered on all channels of
% interest.

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

freq_per_point = span/(length(trace)-1);
start_point = (span-(channel_spacing*num_channels))/2/freq_per_point+1;
points_per_channel = channel_spacing/freq_per_point;

ch_powers=zeros(1,num_channels);    %pre-allocate the array for speed
for channel_num = 1 : num_channels
    temp_trace = trace(round(start_point + (channel_num-1)*points_per_channel)  : ...
                       round(start_point + channel_num*points_per_channel));
    ch_powers(channel_num) = cp(cbw, rbw, span, temp_trace, nbw_cor, rrc_mode, alpha, sym_rate);
end
```

MATLAB function for multi-carrier power.

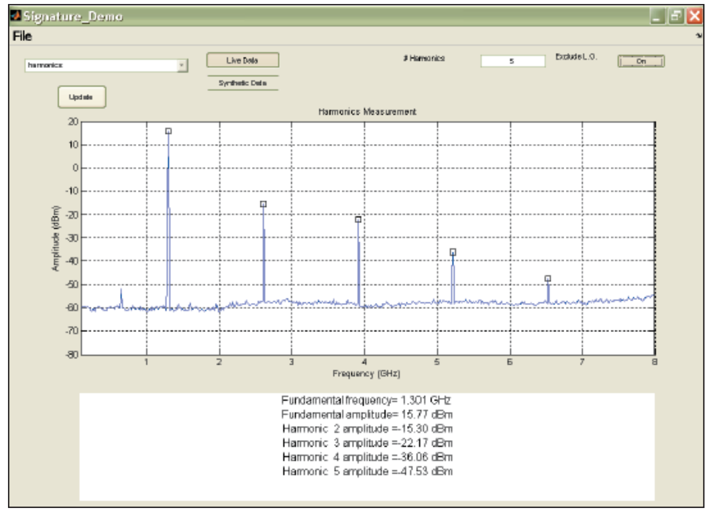
```
function [ch_powers, trace] = mcpnc(cbw, rbw, span, trace, nbw_cor, rrc_mode, alpha, sym_rate,..
    channel_spacing, num_channels, noise_trace)
% multi-carrier power measurement
% ch_powers = mcpnc(cbw, rbw, span, trace, rrc_mode, alpha, sym_rate, channel_spacing)
% ch_powers is a vector of the channel power (in dBm) for each channel in the given trace data
% trace (as an output) is the result of subtracting the noise trace from the input trace
%
% trace (as an input) is assumed to be a 1-D vector of dBm values and span,rbw is the
% associated span and rbw for the trace
% nbw_cor is the noise bandwidth correction value, in linear terms
% rrc_mode can be 'on' or 'off', alpha is the roll-off factor for the RRC
% filter and sym_rate is the symbol rate that is used in the filter
% The RRC filter is applied to each channel
% channel_spacing is the width of each channel
% The spectrum is assumed to be centered on all channels of
% interest.

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

noise_trace=10.^(noise_trace./10);
trace=10.^(trace./10)-noise_trace;
noise_trace=0.01.*noise_trace; %create clip limits 20 dB lower than the noise trace
trace=10*log10(max(trace,noise_trace));

freq_per_point = span/(length(trace)-1);
start_point = (span-(channel_spacing*num_channels))/2/freq_per_point+1;
points_per_channel = channel_spacing/freq_per_point;

ch_powers=zeros(1,num_channels);    %pre-allocate the array for speed
for channel_num = 1 : num_channels
    temp_trace = trace(round(start_point + (channel_num-1)*points_per_channel)  : ...
                       round(start_point + channel_num*points_per_channel));
    ch_powers(channel_num) = cp(cbw, rbw, span, temp_trace, nbw_cor, rrc_mode, alpha, sym_rate);
end
```

MATLAB multi-carrier power function, with noise compensation.

Harmonics

Another common RF measurement is of harmonic content. If we set up the Signature to show the fundamental and the harmonics of interest, we can simply search for the peak value (possibly excluding any LO feed through near dc), and then look at multiples of that frequency. Since the frequency values of the trace points aren't necessarily the exact frequency of the signal, the code has a small search window for each harmonic. This search looks for the highest value in each of the search windows.

Signature's high TOI and low noise floor again allow fast and accurate measurements of harmonics. The instruments TOI specification directly indicates the instrument-generated 3rd harmonic (with no attenuation); you can also use this as a rough indicator of other instrument-generated harmonics.



Measurement of Harmonics.

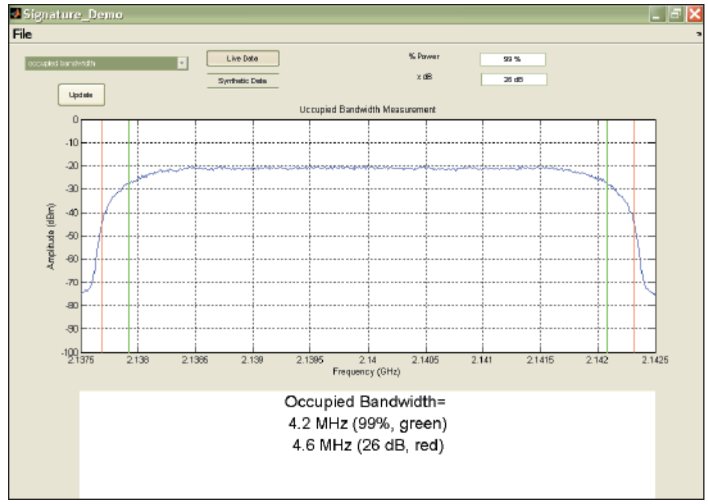
```
function [harmonic_amps,harmonic_freqs] = harmonics(cf, span, trace, num_harmonics, exclude_lo, rbw)
% harmonics measurement
% [fundamental,harmonics] = harmonics(cf, span, trace, num_harmonics, exclude_lo, rbw)
% fundamenal is the frequency of the fundamental, in Hz
% harmonics is a vector of the levels of each harmonic, starting with the fundamental, up to num_harmonics.
% there are num_harmonics + 1 results in harmonics.
% exclude_lo can be 'on' or 'off', and is used to eliminate LO feedthrough when finding the fundamental;
% when exclude_lo is 'on', rbw is used to determine how much of the trace to exclude
%
% cf & span are the analyzer center frequency & span
% trace is assumed to be a vector of dBm values
% num_harmonics is the number of harmonics desired
% exclude_lo ignors the first 10 rbw widths from dc if 'on'
% rbw is used by exclude_lo to determine how much to reject

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

freq_per_point = span/(length(trace)-1);
start_freq = cf - span/2;

if strcmp(exclude_lo,'on')
    exclude_freq = rbw * 10;                            % 10 * rbw is > 100 dB rejection
    exclude_points = (exclude_freq - start_freq) / freq_per_point;
    exclude_points = max(ceil(max(exclude_points , 0)),2);  % eliminate cases where the start frequency
                                                        % is greater than the exclude frequency
                                                        % and always exclude the 1st 2 points
    trace (1:exclude_points) = -200;                    % set points in the lo exclusion range below any trace
end

peak_location = find(trace==max(trace));                % find the location of the maximum point
peak_location = peak_location(1);                       % if multiple points at same level, pick 1st
fundamental = start_freq + (peak_location-1) * freq_per_point;  % compute fundamental frequency from point #
harmonic_amps(1)=trace(peak_location);                  % store the amplitude in the result array
harmonic_freqs(1)=fundamental;                          % store the frequency in the result array

if fundamental>0
    for harmonic_num=2 : num_harmonics
        location_of_harmonic = round(((fundamental * harmonic_num) - start_freq) / freq_per_point + 1);
        if location_of_harmonic <= length(trace) - harmonic_num
            start_search_loc=max(location_of_harmonic - harmonic_num , 1);
            stop_search_loc=min(location_of_harmonic + harmonic_num , length(trace));
            search_trace = trace(start_search_loc  :  stop_search_loc);
            peak_location = find(search_trace==max(search_trace));  % find largest point near harmonic location
            peak_location = peak_location(1);                       % if multiple points at same level, pick 1st
            harmonic_amps(harmonic_num) = trace(location_of_harmonic - harmonic_num + peak_location -1);
            harmonic_freqs(harmonic_num) = start_freq + (location_of_harmonic + ...
                peak_location - harmonic_num - 2) * freq_per_point;
        end
    end
end
```

MATLAB function for measuring harmonics.

Occupied Bandwidth

The Occupied Bandwidth (OBW) measurement shows the frequency range that contains a percentage (usually 99%) of the entire energy in the measured span. For this to be meaningful, a measurement span must also be specified.

A related measurement, sometimes called emission bandwidth, shows the frequency range that contains amplitudes above a particular level— usually 26 dB below the signal peak.



Occupied Bandwidth measurement.

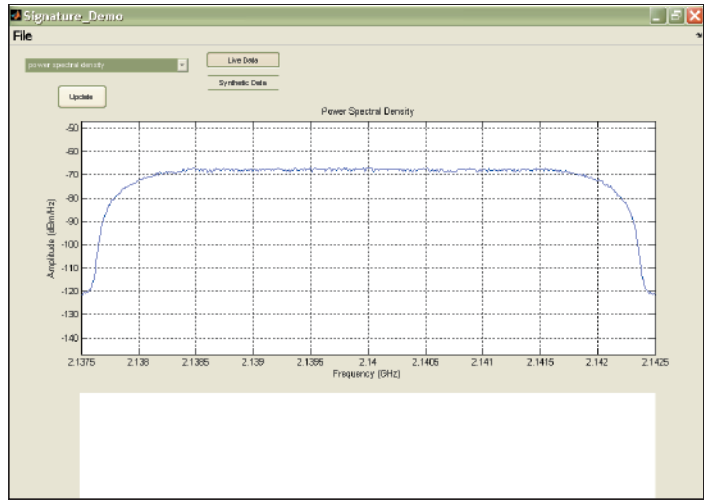
```
function [percent_bw, xdB_bw, freq_offset] = obw(span, percent_power, xdB, trace)
% [percent_bw, xdB_bw, freq_offset]  = obw(span, percent_power, xdB, trace)
% calculates and returns the xdB (in dB) bandwidth and the bandwidth
% occupied by percent_power (in %) for the data in 'trace' which
% has a frequnecy span of 'span'
% freq_offset is the difference between the center & the half-power point(in Hz)


% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

% convert trace to Watts
 y = 1e-3 * 10.^(trace/10);
 total_power = sum(y);

% find the half-power point in the trace
integ_power = 0;
half_power_point=0;
while integ_power < total_power/2
    half_power_point = half_power_point + 1;
    integ_power = integ_power + y(half_power_point);
end

% frequency offset calculation
freq_per_point = span / (length(trace)-1);
freq_offset = (half_power_point - ceil(length(trace)/2)) * freq_per_point ;

% obw calculation - Start from half-power point and integrate out until percent_power is reached
temp_power = 0;
left_obw_location = 0;
while (temp_power/total_power) < ((100-percent_power)/200) % 200 because left half of 100%
    left_obw_location = left_obw_location + 1;
    temp_power = temp_power + y (left_obw_location);
end
temp_power = 0;
right_obw_location = length(y) + 1;
while (temp_power/total_power) < ((100-percent_power)/200) % 200 because right half of 100%
    right_obw_location = right_obw_location - 1;
    temp_power = temp_power + y (right_obw_location);
end
percent_bw = (right_obw_location - left_obw_location) * freq_per_point;

%xdB calculation -
delta_power = trace - max(trace);   %normalize to the peak to simplify search
left = 1;
while delta_power(left) < (-xdB)    % find left x dB point
    left = left+1;
end
right = length(trace);
while delta_power(right) < (-xdB)   % find right x dB point
    right = right-1;
end
delta = right-left;
xdB_bw = freq_per_point * delta;
```

MATLAB Occupied Bandwidth (OBW) function.

Power Spectral Density

To convert a trace to power spectral density (PSD), use the RMS detector and offset the trace by 10*log(RBW), plus the same correction factor for the noise bandwidth as used in the channel power measurement. Use the RMS detector for making PSD measurements.

Note that the name of this function is `Signature_psd`, as there is a MATLAB function already named `psd` (in the Signal Processing Toolbox).



Power Spectral Density (PSD) measurement.

```
function [psd_trace,psd_ref_level]=Signature_psd(trace,rbw,ref_level,nbw_to_rbw)
% function [psd_trace,psd_ref_level]=psd(trace) converts trace to power spectral density
% trace is a vector that describes a spectrum, in dBx
% rbw is the instrument resolution bandwidth
% ref_level is an optional input of the instrument's reference level.  If
% input, the psd_ref_level output is computed with the same changes as the
% trace
% nbw_to_rbw is an optional correction factor that converts the resolution
% bandwidth into noise bandwidth (in linear terms, not dB)
% note: must use rms detector on instrument to get correct answers

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

if ~exist('nbw_to_rbw','var')
    nbw_to_rbw=1;
end


psd_trace=trace-10*log10(rbw*nbw_to_rbw);

if exist ('ref_level','var')
    psd_ref_level=ref_level-10*log10(rbw*nbw_to_rbw);
else
    psd_ref_level=NaN;
end
```

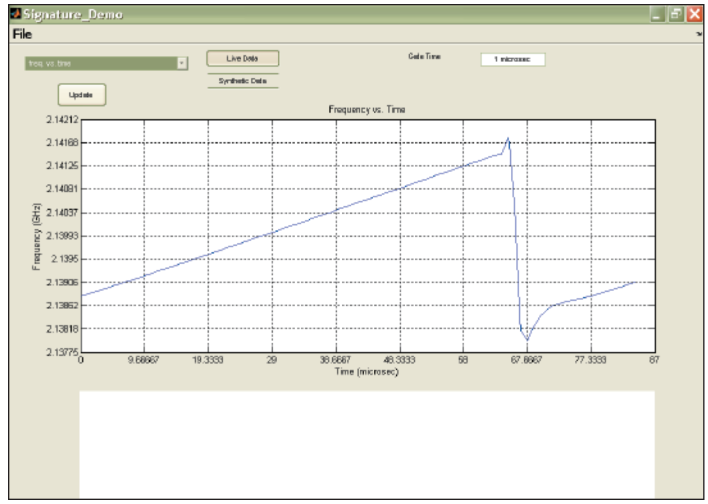
MATLAB power spectral density (psd) function for Signature.

Frequency versus Time

The display of frequency versus time is very useful when looking at the transient response of Voltage-Controlled Oscillators (VCOs), Direct Digital Synthesizers (DDSs), and Phase-Locked Loops (PLLs). With the IQ vectors from Signature, it's easy to compute frequency vs. time—just use the angle function to convert to phase, then compute the phase change versus time. The MATLAB unwrap function is essential to this process. This function detects a large phase change (close to 360°) between adjacent points, and adds in the "missing" 360°.

The gate time parameter lets you trade off frequency resolution versus time resolution. The longer the gate time you choose, the better the resulting frequency resolution, but the poorer the time resolution.

Note that the frequency resolution of this technique is extremely high; the limiting factors are only the signal-to-noise ratio and the phase noise of the Signature local oscillator. The excellent phase noise performance of Signature allows frequency measurement capability exceeding the best frequency counters available.



Frequency versus time display of a chirp signal.

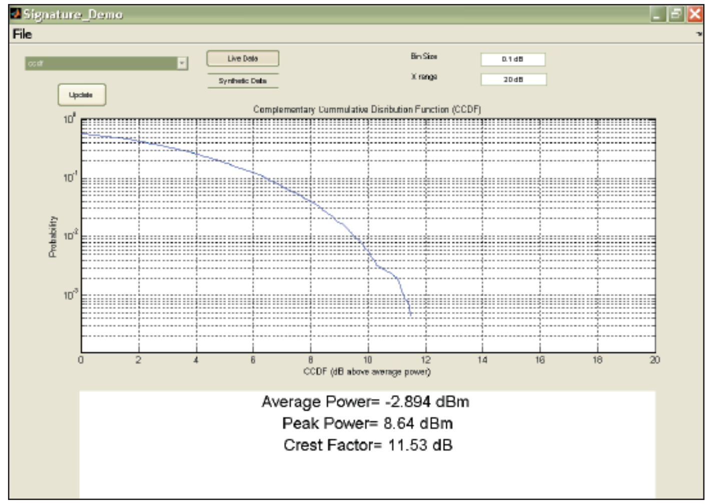
```
function [freqs, times, gate_time] = FvsT(IQ, cf, sample_period, gate_time)
% function [freqs,times,gate_time] = FvsT(IQ, sample_rate, gate_time)
% Converts I/Q vectors into frequency versus time
% IQ is a vector of complex points, each representing a point in the IQ
% plane.
% cf is the center frequency of the analyzer.
% sample_period is the the time between samples, in seconds.
% gate_time allows the frequency to be integrated over several
% sample points.
% freqs is a vector with the frequency values
% times is a vector of the start of each gate interval.

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

% Compute gate time
gate_points=round(gate_time/sample_period);
gate_time=gate_points*sample_period;
num_gates=floor(length(IQ)/gate_points);

% preallocate memory
freqs=zeros(num_gates,1);
times=zeros(num_gates,1);

% Compute phase
Phi=unwrap(angle(IQ))/(2*pi);    %In cycles

%Compute frequency
for i=1:num_gates;
    Phi_start_index=(i-1)*gate_points+1;
    Phi_stop_index=round((i)*gate_time/sample_period)+1;
    freqs(i) = (Phi(Phi_stop_index)-Phi(Phi_start_index))/gate_time;    %F=dPhi/dT
    freqs(i) = cf + freqs(i);            %Add center frequency to get F vs. T at RF
    times(i) = (i-1) * gate_time;
end
```

MATLAB function to convert IQ vectors into frequency versus time.

CCDF

The Complementary Cumulative Distribution Function (CCDF) is used to understand the amplitude statistics of modulated signals. The graph shows the probability that a peak exceeds the average amplitude by a number of dB. The graph at the right shows an example CCDF plot.

CCDF is easy to compute by using the MATLAB `hist` function on the signal power to get the histogram, then integrating the histogram to get the CCDF curve.



Example CCDF plot.

```
function  [count,x_axis]=ccdf(power, bin_size, x_range)
% ccdf, complementary cummulative distribution function
% [count,x_axis,total_count]=ccdf(power, bin_size, x_range)
% Converts vector of Power into a CCDF plot
% This shows the probability that the signal is above the
% mean power.
%
% Outputs:
% count- a vector with the ccdf curve
% x-axis - a vector that shows the bins for the ccdf
%
% Inputs:
% power - a vector with power values, in dBx
% bin_size - the ccdf x-axis resolution, default 0.1 dB
% x_range - the maximum x-axis value, default 20 dB

% Copyright 2004 Anritsu Company
% Revision 1.2 4 November 2004

% Set default values for bin_size & x_range if not defined
if nargin<2
    bin_size=0.1;
end
if nargin<3
    x_range=20;
end

average_power=mean(power);                                          % Find the histogram starting point
x_axis=(average_power-0.1 : bin_size : average_power+x_range);     % Create histogram bins
count=hist(power,x_axis);                                           % Histogram the Power

total_count=sum(count);      % Count all points to enable proability calculation
count=count(2:end);          % Eliminate points below average power
x_axis=x_axis(2:end);        % Eliminate points below average power

count=fliplr(cumsum(fliplr(count)));    % Integrate histogram from the right (higher amplitude) for CCDF
count=count/total_count;                % Convert integrated histogram to probability
```

MATLAB code for creating the CCDF data.

Since the histogram is done on the power, the CCDF could be computed using a zero-span trace, but this is limited to only 501 points.

The IQ vectors in Signature can provide many thousands of points instead of just 501.

Before using the CCDF function shown here, you must convert the IQ vectors to power by using the MATLAB `abs` function, and then convert to dB, e.g.:

```
Power=20*log10(abs(Signature_IQ_Data))
```

To plot the computed CCDF curve, use the MATLAB `semilogy` graphing function:

```
Average_power=mean(Power);
Total_count=length(Power);
[Count, X_axis]=ccdf (Power, Bin_size, X_range);
semilogy(X_axis-Average_power,Count)
axis([0,X_range,1/Total_count,1]);
```

Spectrograms

A spectrogram is a display that shows the 3 dimensions of amplitude, frequency, and time, all on a single plot. It does this by showing amplitude using color, and using the other axes for frequency and time.

There are several ways to compute a spectrogram using MATLAB, including the `specgram` function, the `specgramdemo` function, as well as manually computing the FFTs and building the display.

Specgram function

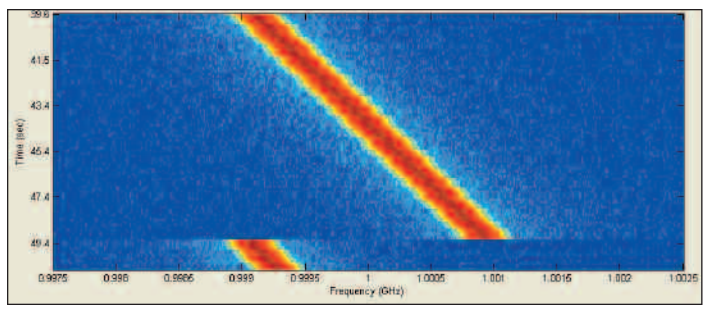
If you have installed the Signal Processing Toolbox option to MATLAB, you can use the specgram function. This will compute a spectrogram from the IQ data, including FFT windowing, and display the result. Here are a few examples of using the `specgram` function:

Use all defaults:

```
specgram(Signature_IQ_Data)
```

Change the FFT size to 1024, specify that the sampling rate is 20 ns, so the time axis is correct:

```
specgram(Signature_IQ_Data, 1024, 20e-9)
```



MATLAB `specgram` display of a chirp signal, including correct time & frequency axes, and rotation to show Frequency on the x-axis.

You can also specify different windows and use overlapping to improve the time resolution.

The presentation of the MATLAB `specgram` function, however, has two limitations. First, the axes are swapped compared to the traditional instrument presentation—while instruments usually have frequency on the X-axis, `specgram` has frequency on the Y-axis. Second, the frequency range for `specgram` for IQ data is from 0 Hz to the sampling frequency (F_S), while the IQ vectors in Signature range from $-F_S/2$ to $F_S/2$. These limitations are easy to fix by using the MATLAB transpose operator (a single quote), `fftshift`, and scaled image (`imagesc`) functions:

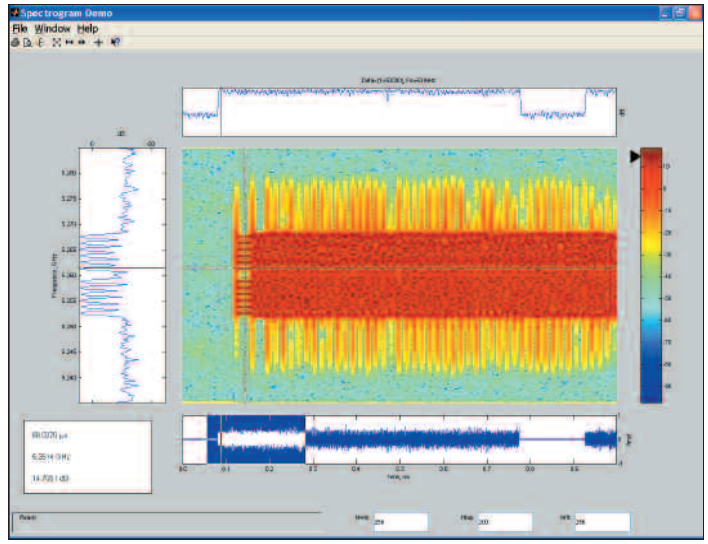
```
Y=20*log10(abs(fftshift(specgram(Signature_IQ_Data),2))');
imagesc(Y)
```

Labeling the Spectrogram Axes

The figure on the previous page has the X-axis labelled with the analyzer center frequency. The normal labelling of the specgram axes are related to the sampling time. You can manually label the axes as well, by using the following commands:

```
set(gca,'XTickLabel',label_string)
set(gca,'YTickLabel',label_string)
```

The label_string can be in a variety of formats, such as a string array. Check the MATLAB help documentation on Axes Properties for details.



Signature_specgramdemo *result on an 802.11a signal.*

MATLAB Spectrogram Demo

The Signal Processing Toolbox in MATLAB also includes a more advanced spectrogram display, called specgramdemo. This includes various display additions, including a time overview as well as "slices" of time and frequency delimited by markers. The figure shows the results of running specgramdemo on a chirp (frequency sweep) signal.

Signature Option 40 includes a modified version of specgramdemo that shows the spectrum of the correct frequency range, including the Center Frequency on Signature.

You can call this with the following line: Signature_specgramdemo(double(Signature_IQ_Data),... 1/Signature_Setup_Data.Sampling_period)

Building your own Spectrogram

A third way to get a spectrogram is to create it from scratch. You can build a matrix using multiple FFTs, and then display the spectrogram by using imagesc. This allows, for example, building up a spectrogram from multiple acquisitions. For example, assuming Signature is in FFT mode:

```
Y=Signature_Trace1;
for i=1:100
      Y=cat(1,Y,Signature_Trace1);
      pause(0.1);
end
imagesc(Y);
```

The pause statement allows the instrument to take new data. You may need a longer pause if you are using very narrow resolution bandwidths. You can also use handshaking instead of the pause statement; this will work with any RBW.

Spectrograms from IQ vectors vs. from Traces

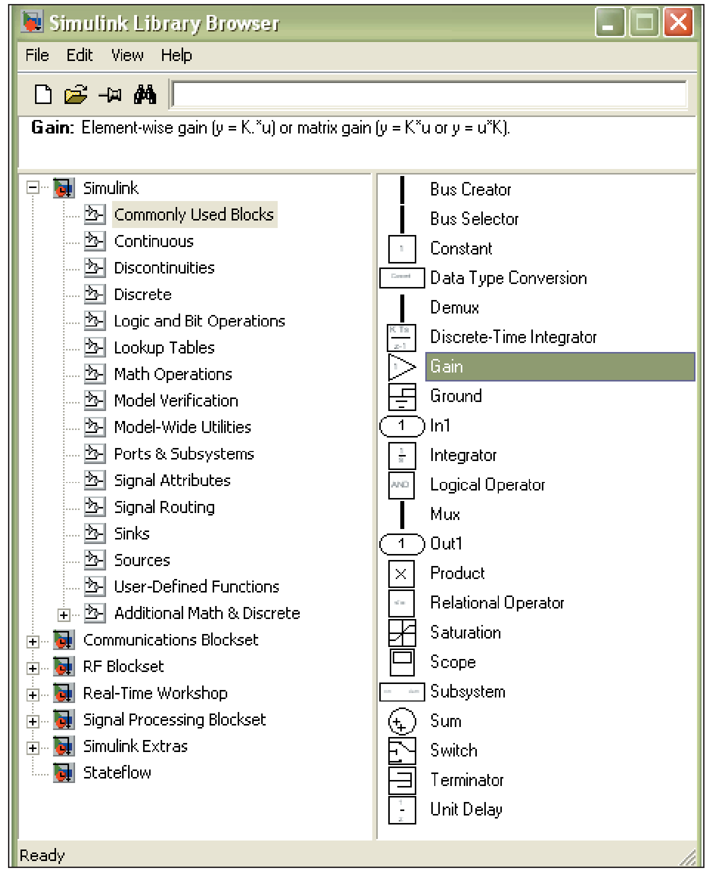
As we have seen in the above examples, there are 2 different methods of building a spectrogram. Using the IQ vectors provides continuous information over a short time frame (up to about 1 second). Using instrument traces provides a much longer time – of minutes, hours, or even days.

Using Simulink

Simulink is another product from The MathWorks that has advantages for developing demodulation models. Simulink uses a block-diagram-editor and deals with time as a simulation parameter.

The figure shows the Simulink Library Browser. This is where you can find blocks and add them to your model.

This section will show you how to get Signature information into Simulink, perform a simple demodulation, make some measurements, and return the results to MATLAB.



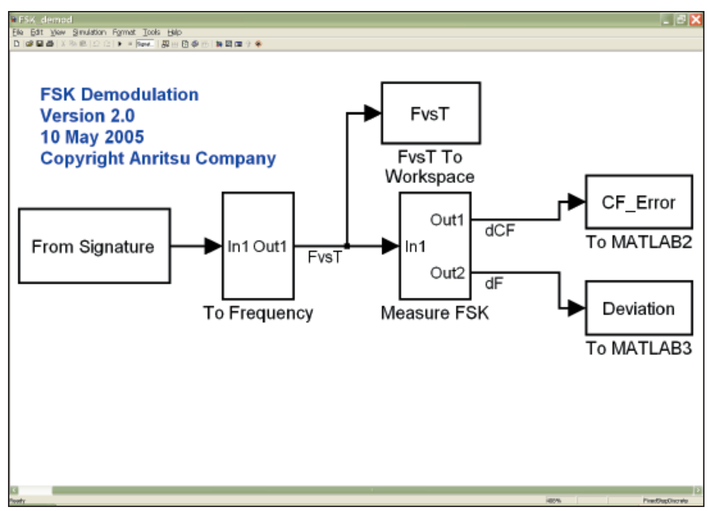
Simulink Library Browser.

FSK Demodulation

The figure shows an example Simulink model for demodulating and measuring a signal that uses Frequency-Shift-Keying modulation.

The blocks in the model are:

- `From Signature`
  This is a Simulink Signal from Workspace block, which lets you get data from MATLAB (and therefore Signature), and set the sample rate. Note that this block is from the MATLAB Signal Processing Blockset.
- `To Frequency`
  This is a Simulink subsystem, which contains several other blocks. At this level, it is a simple FSK demodulator—it converts baseband IQ vector data into frequency-versus-time data.
- `Measure FSK`
  This is another subsystem, which takes the frequency-versus-time values and measures the center frequency error and the deviation.
- Three "`To MATLAB`" blocks.
  ("`FvsT To Workspace`", "`To MATLAB2`", and "`To MATLAB3`")
  These are Simulink "`To Workspace`" blocks, which make the measurement results available to MATLAB.

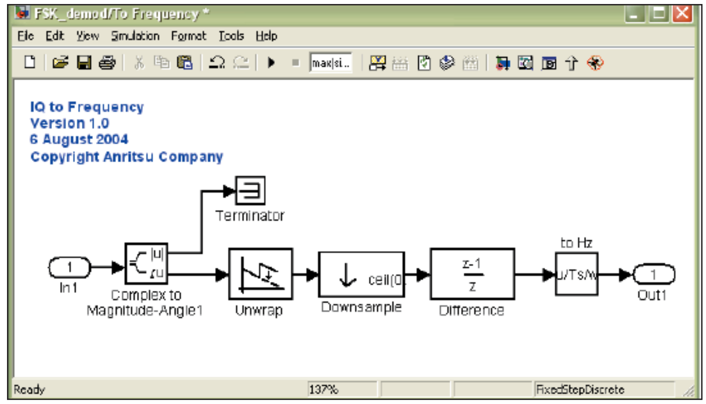


Simulink FSK demodulation & measurement block diagram.

“To Frequency” Block

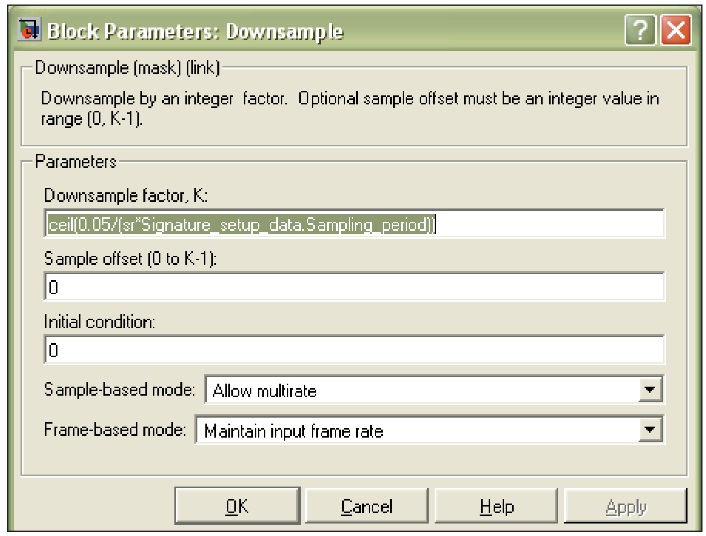
The conversion to frequency subsystem:

- Converts the IQ vectors to phase
- “Unwraps” the phase. This means that sharp transitions are eliminated, which allows the phase to be more than 360 degrees
- Reduces the sampling rate to be twice the symbol rate
- Takes the difference between adjacent phase readings
- Converts the phase changes to frequency. This conversion is based on the phase change and the sample rate.



Simulink “To Frequency” subsystem converts IQ vectors into frequency versus time.

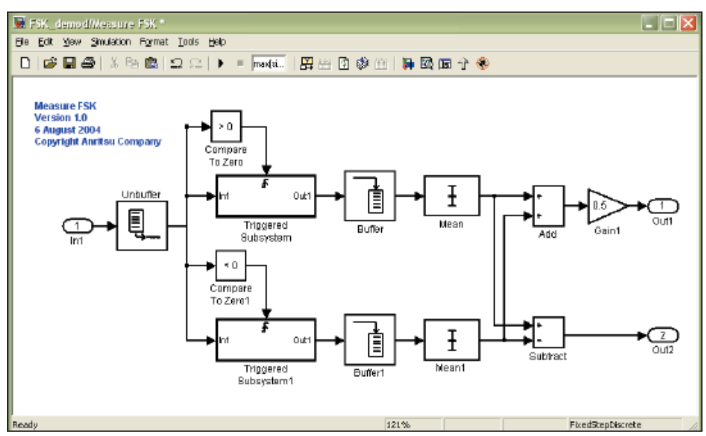
If you double click on the Downsampling block, you will see that there is an equation to select the downsample factor. This equation is based on the sampling rate from Signature and the expected symbol rate of the signal. You must specify the expected symbol rate in a MATLAB variable named `'sr'`. The equation shown gives 20 samples per FSK signal.



Simulink parameters for the Downsample block can use MATLAB variables.

“Measure FSK” block

This subsystem sorts the frequency data into 2 groups – those above the center frequency (which is zero Hz at baseband), and those below the center frequency. Each of these groups is then averaged to get an estimate of the high frequency and low frequency states. These two frequency estimates are then averaged to get a measurement of the center frequency error (Out1); the difference is taken to get the frequency deviation (Out2).



“Measure FSK” subsystem sorts frequency measurements into 2 groups to determine center frequency error & deviation.

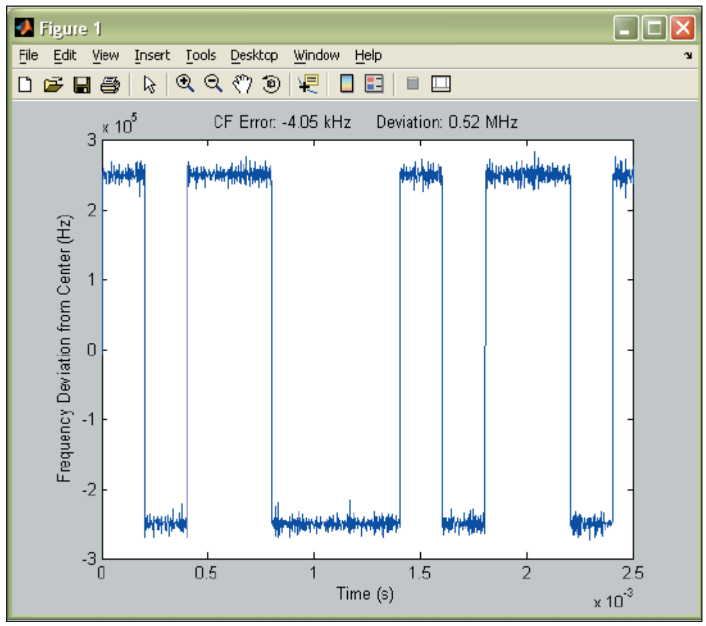
Getting MATLAB data into Simulink

The Signature IQ vectors in MATLAB must be reformatted before they are transferred to Simulink. This is because Simulink requires complex inputs to be in a structured format. The following code shows how to do this.

```
%sim_prep--convert Signature IQ vectors into format for Simulink

% Copyright 2004 Anritsu Company
% Revision 1.0 28 July 2004

IQ_length=length(Signature_IQ_Data);
sample_time=Signature_Setup_Data.Sampling_period;
simin.time=(0:sample_time:(IQ_length-1)*sample_time)';
simin.signals.values=Signature_IQ_Data';
simin.dimensions=[1];
```

Measurement Results

The figure to the right is a MATLAB graph that shows the frequency-versus-time measurements, as well as the resulting center frequency error and frequency deviation.



Instrument Control

Controlling Signature through Web Services

In addition to a GPIB interface, Signature can also be controlled by a Web Services connection. This is available both via the LAN, as well as directly on the instrument. This means that you can also use MATLAB to control Signature measurements right on the instrument.

The figure shows a simple example of using MATLAB to control Signature via the Web Services interface. It sets up the Web Services (if they aren't set up already), presets the instrument, then sets the center frequency to 1 GHz.

If you want to use this code to talk to Signature via a network, instead of on the instrument, change the `host` from "localhost" to the Instrument Name. You can find the Instrument Name by going to the System menu, selecting Configuration, IO Config, and then Instrument Name.

```
%WebServicesExample
%Preset Signature & set Center Frequency to 1 GHz using the Web Services interface

% Copyright 2005 Anritsu Company
% Revision 1.0, 24 June 2005

host = 'localhost';     % Signature hostname.

% Set up a web services object (named 'spa') for spectrum analyzer controls
url = ['http://' host '/SignatureSpectrum/SignatureSpectrum.asmx?wsdl'];
createClassFromWsdl(url);   %Creates '.m' files in the folder @SignatureSpectrum
                            %in the current directory
spa = SignatureSpectrum;    %Creates an object that refers to the Signature Spectrum analyzer
                            %web services, at the address of 'host'

% setup a web services object (named 'sys') for system controls
url = ['http://' host '/SignatureSystemControl/SignatureSystemControl.asmx?wsdl'];
createClassFromWsdl(url);   %Creates '.m' files in the folder @SignatureSystem
                            %in the current directory
sys = SignatureSystem;      %Creates an object that refers to the Signature System
                            %web services, at the address of 'host'

Preset(sys)                         %Preset Signature
SetCenterFrequency(spa,1,'GHz');    %Set Signature Center Frequency to 1 GHz
```

Use the Web Services interface to control Signature from MATLAB running on the instrument or another computer.

You can get the list of Web Services commands on Signature several ways:

- The Signature programming manual. This is available both as a printed manual, and through the Documentation item in the Help pulldown menu.
- After you run the CreateClassFromWsdl command in MATLAB, there will be a new directory created under the current directory (usually C:\Signature\MathWorksConnectivity). You can see what commands are available by using MATLAB to look at the files in these directories. Since Signature has 3 Web Services, there can be 3 directories, called:
  - @SignatureSystem
  - @SignatureSpectrum
  - @SignatureModulation
- Use a web browser to look at the web descriptions of the available services. You can look at the available commands, see the syntax for each command, and in many cases test the operation of the command. In a web browser running on the instrument, look at:
  - http://localhost/signaturesystemcontrol/signaturesystemcontrol.asmx
  - http://localhost/signaturespectrum/signaturespectrum.asmx
  - http://localhost/signaturemodulation/signaturemodulation.asmx

One of the benefits of the Web Services interface is that it is "location independent". The example code is written to run directly on Signature, but by changing the definition of the `host` variable in the example program, you can control a Signature connected to the network. Note that the host name is embedded in a file that MATLAB creates when you use the `createClassFromWsdl` function.

The `createClassFromWsdl` function does have a fair amount of overhead (5-10 seconds for SignatureSpectrum and about 1 second for SigantureSystem), so you may not want to use it every time you run your code. As long as you are referring to the same instrument, this works fine.

Note that there is a bug in the Web Services implementation in MATLAB R14SP2 that causes problems when using Signature. You can go to MATLAB Central to get patches to fix this bug at:
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=7938&objectType=FILE
Versions of MATLB beyond R14SP2 are expected to include these patched files.

Also note that there is a short delay (typically about 20 ms) after making a Web Services call before `Signature_Setup_Data` is updated.

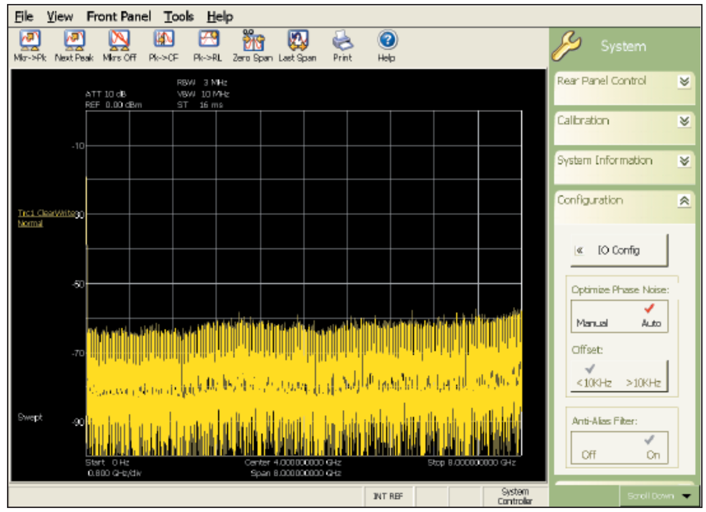
You can also read data from Signature via Web Services. For example, to read Trace 1 into a variable called Trace1, use the following code (after using `CreateClassFromWsdl` & `spa=SignatureSpectrum` as shown above).

```
Trace=GetTraceData(spa,1);
str2num(char(Trace.float));
```

GPIB Control of Other Instruments from MATLAB

The Instrument Control Toolbox in MATLAB allows controlling instruments through GPIB and other interfaces. If you have the GPIB interface in Signature (Option 3), you can easily control other instruments, such as signal sources. You need to set Signature to be the GPIB System Controller by:

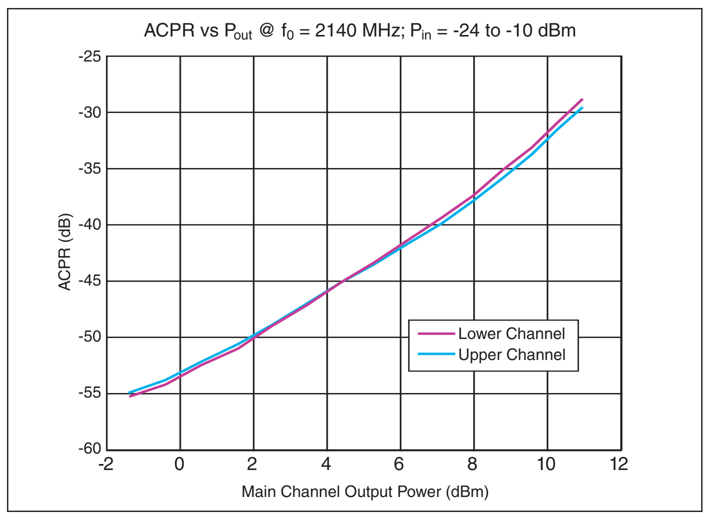
- Selecting System, Configuration, IO Config, GPIB.
- In the National Instruments Measurement & Automation Explorer that comes up, click on 'Devices and Interfaces', right click on GPIB0, and select Properties.
- Click on the System Controller box, click OK, and close the Measurement & Automation Explorer.
- On the lower right corner of the display area, Signature shows that it is now in System Controller mode.



You can tell when Signature is the GPIB System Controller mode by the annotation in the lower right corner of the display area.

Example of Controlling Instruments—Measuring ACPR versus Power

An example of using MATLAB to control both other instruments via GPIB (using the MATLAB Instrument Control Toolbox), and Signature using Web Services is available. The example uses an Anritsu MG3700A Vector Signal Generator to generate a WCDMA modulated signal, and then measures Adjacent Channel Power Ratio of a Device as the input power level is changed. You can look at the example in the file `ACPvsP`.



ACPR vs. Output Power of a class-A amplifier using the `ACPvsP` script.

MATLAB Demonstration

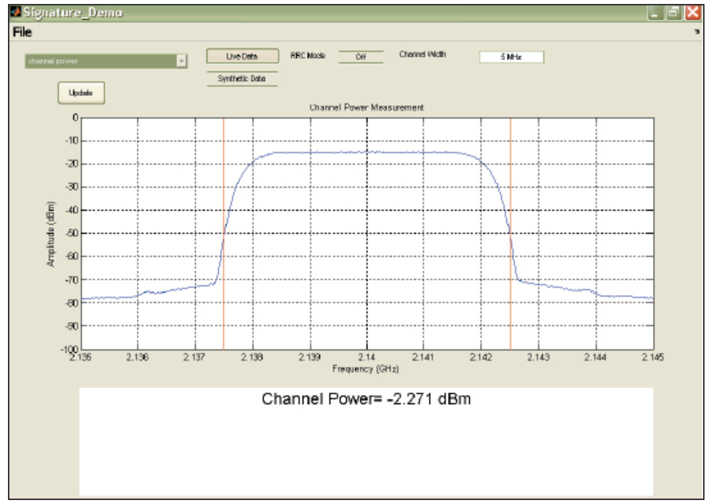
The Signature "Connectivity to MATLAB" option (Option 40) includes an example graphical user interface that uses many of the above functions and automatically updates measurements using timers. An example of this demonstration doing a channel power measurement is shown in the figure.

To invoke this example, type the following on the MATLAB command line:

```
Signature_Demo
```

If you wish to use a different trace than Trace 1, or start the demonstration using a specific measurement, you can use the longer form of the command, shown below. Note that the last parameter is optional and is the name of one of the items in the dropdown box in the upper left corner of the demo GUI.

```
Signature_Demo Signature_Trace1...
Signature_Setup_Data Signature_IQ_Data...
'channel power'
```



MATLAB demonstration GUI for Signature.

Using the MATLAB Demonstration

The various demonstrations use different data from Signature. Most of the demonstrations use the spectrum output. The "plot trace" demo works with zero-span data. The CCDF and Frequency vs. Time demonstrations require IQ vectors from Signature. To do this, make sure that the IQ vector output is turned on in the MATLAB setup dialog and put the instrument into Modulation Measurement mode.

The spectrogram works on either spectrum data or on IQ vectors. If spectrum data exists, a spectrogram is built out of successive spectra. If spectrum data doesn't exist, but IQ data is available, the spectrogram is built from the IQ data; this allows a much shorter time span for the measurement. To get IQ data, the instrument must be in the Modulation Measurements mode.

Note that once a spectrum is output to MATLAB it is never cleared by Signature. So if you want a spectrogram from IQ data, you must either start the MATLAB interface with the Trace output turned off, or after switching the instrument to Modulation Measurements mode, you would type the following line of code in the MATLAB command window:

```
clear Signature_Trace1
```

How to get support

Anritsu and The MathWorks are both committed to supporting you using MATLAB on Signature. For support on Signature connectivity to MATLAB, contact Anritsu by going to www.anritsu.com, clicking on “Contact Us”, and selecting your country. This will list your local phone number and e-mail address. In the U.S., you can also call 1-800-ANRITSU (267-4878).

For support on the details of MATLAB, contact The MathWorks via their web site.
Go to www.mathworks.com and click on “Support”.

Conclusion

By combining MATLAB with the Anritsu Signature High Performance Signal Analyzer, you can do your own analysis quickly and easily right on the instrument, including live updates of traces and measurements.

References

1. The MathWorks
3 Apple Hill Drive, Natick, MA 01760-2098
Phone: 508-647-7000; Fax: 508-647-7001; http://www.mathworks.com

SALES CENTERS:
United States (800) ANRITSU
Canada (800) ANRITSU
South America 55 (21) 2527-6922
Europe 44 (0) 1582-433433
Japan 81 (46) 223-1111
Asia-Pacific (852) 2301-4980
Microwave Measurement Division
490 Jarvis Drive, Morgan Hill, CA 95037-2809
http://www.us.anritsu.com

Discover What's Possible®

11410-00353 Rev. B