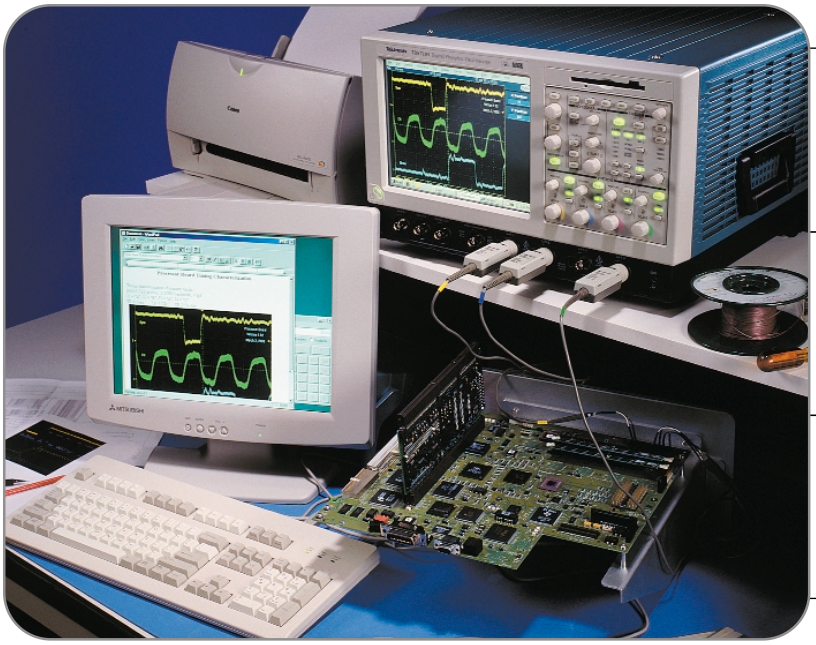
# Analyzing Clock Jitter Using MATLAB





## Easy data connectivity using the TDS 7000 Series.

Characterizing jitter, whether in the 100 picoseconds (ps) or 100 microseconds (µs) range, can be a time consuming effort requiring detailed analysis of waveform data. Automating this analysis can result in increased accuracy, improved efficiency and significantly improved test repeatability as compared to a manual process.

The TDS7000 Series digital phosphor oscilloscopes, with open Windows®, offers the unique advantage of enabling industry-standard analysis and data-base applications, such as Excel, Mathcad® and MATLAB®, to reside within the instrument itself.[1] The waveform data in the oscilloscope's acquisition memory can be exported, then imported to the analysis application, processed and displayed—all on the same platform.

This document will describe the process of using the TDS7000 Series digital phosphor oscilloscopes and MATLAB to capture signal data and then to conduct simple jitter analysis of an NRZ (non-return-to-zero) clock signal. This includes:

- How to determine the sample resolution for accurate results
- How to move waveform data from the TDS7000 to MATLAB
- How to build a simple jitter analysis worksheet
- How to use MATLAB to graph the results

[1] The TDS7000 supports a second Windows monitor output which will greatly enhance the oscilloscope's functionality as an analysis system. The second monitor is not required but enables you to actively use the scope while simultaneously running a separate application on the second monitor.



Tektronix®

## Overview of Jitter

### What is Jitter?

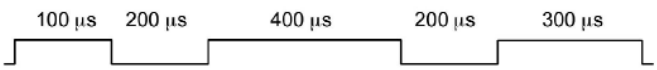
Jitter has two overlapping definitions:

1. The deviation of a signal transition from its ideal position in time, or…
2. The timing variation from transition to transition

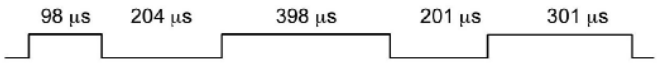
Jitter intervals often fall in the range from tens of ps to a few hundreds of ps. As clock frequencies reach the 1 GHz range, this seemingly small amount of jitter error can become a significant portion of the "timing budget," that is, the time allotted for a series of logical operations. For example, at the standard SONET/SDH bit rate of 2.5 Gbits/sec, one unit interval (one data bit) is only 400 ps. The transmitter and receiver components consume most of this budget. Jitter can make too much of the remaining time unavailable (or at least uncertain) for necessary operations.

### Jitter Characteristics

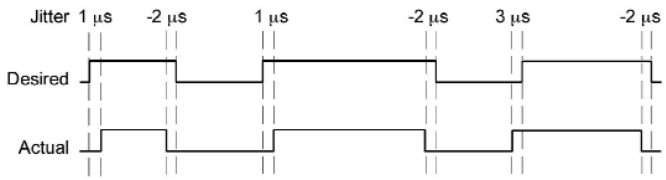
Let's take an illustrated look at jitter. Consider the following waveform:




Every time the waveform goes positive above a specified threshold, the data is a logical one. When the waveform is below the threshold, the data is a logical zero. Now suppose that when this waveform is sent and received, its timing is affected as shown below:

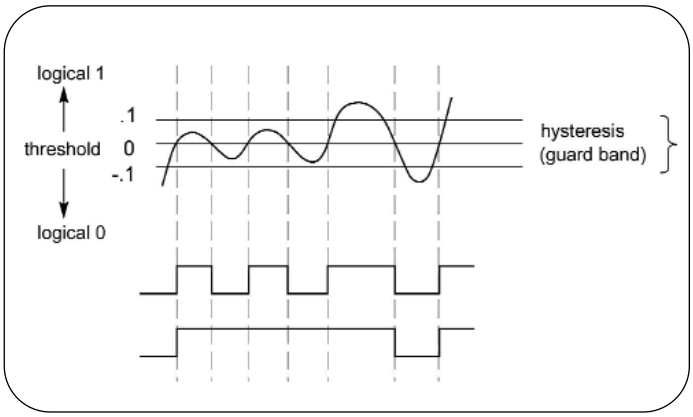



Here the signal transition deviates from its ideal position in time and it varies from transition to transition—the definition of jitter. Where the location of the edges has changed, because of factors such as noise or other error sources, the amount of change is the jitter figure, expressed in microseconds (µs), nanoseconds (ns), or picoseconds (ps), depending on the magnitude of the jitter.




### Hysteresis Effects and Guard Bands

When detecting and evaluating jitter, threshold crossings and hysteresis must be considered. It is common practice to use a "guard band" (essentially a margin that widens the ideal threshold). Figure 1 shows the threshold to be crossed and the guard band around it. To avoid registering noise as an actual change in logic level, a threshold crossing is not considered a valid transition unless the waveform first goes outside the guard band. Enforcing this rule ensures that only the true edges (bottom trace in Figure 1) are detected.




▶ ***Figure 1:*** *The threshold and hysteresis in clock jitter analysis*

### Jitter and Your Design

It's an acknowledged fact in the industry: jitter can affect the stability of any high-speed circuit design. Increasingly, this realization is reflected in standards, specifications and compliance guidelines.

A case in point is the USB 2.0 Specification promulgated by the USB Implementer's Forum. This document exists to ensure interoperability among USB-based products from diverse manufacturers and, like other standards, is essential to the continued success and health of the platform. Consumers will demand USB 2.0 capability as the amount of readily accessible multimedia content grows in the coming years.

USB 2.0 has a data rate of up to 480 Mbits/sec., which is sufficient to deliver video and other bandwidth-intensive content. It is also fast enough to be susceptible to jitter. Therefore the Compliance Suite for USB 2.0 Certification includes jitter specifications. Any new platform that includes USB 2.0 (an increasing number of consumer and office products) must meet these guidelines in order to qualify for the certification.

The USB 2.0 Specifications are just one example that demonstrates how a few ps of timing inaccuracy can affect not only the technical functionality, but also the market viability of a new product.

### The Algorithm

The algorithm we'll use is as follows:

- Measure the timing of the edges
- Derive the clocks using the edges and the symbol rate
- Determine the average measured symbol rate
- Calculate the error in the average measured symbol rate
- Reconstruct the timing of the edges using the derived clock and the average measured symbol rate
- Calculate the jitter from the measured timing of the edges and the reconstructed timing
- Plot the jitter

## Terminology

The clock jitter example will make use of the following terms:

Table 1: Terminology used in the clock jitter problem

| Term | Meaning |
|---|---|
| Symbol Rate | The frequency at which the communication system is sending data. (In RS232 this would be the baud rate.) |
| Sample Rate | The frequency at which the oscilloscope is sampling data (measured in samples/second). |
| Sample Interval | The time difference between samples (measured in seconds/sample). The oscilloscope user interface calls this resolution. Mathematically the *Sample Interval = 1/ Sample Rate*. |
| Threshold | The voltage used to determine if a voltage value is a logical zero (0) or a logical one (1). |
| Edge | A place where the waveform crosses the threshold. We are interested in the time when this occurs. Since an edge usually occurs between samples, we must use a technique called linear interpolation to find it accurately. |
| Hysteresis | Defines a guard band around the threshold that is used to make the algorithm less sensitive to small amounts of noise. For each edge there must be at least one point outside the guard band. |

## Determining the Best Sample Interval

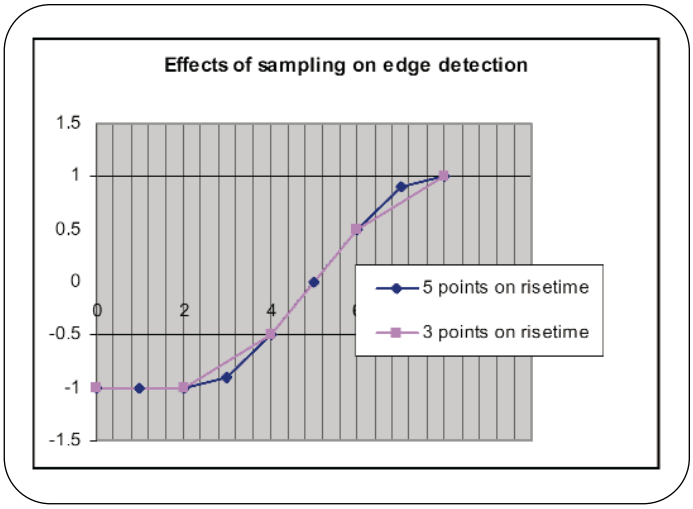
Before acquiring the waveform data, it is necessary to decide on the best sample interval to use. In trying to measure jitter, we must balance two things:

- Capturing as many edges as possible (or desired)
- Locating each edge as accurately as possible

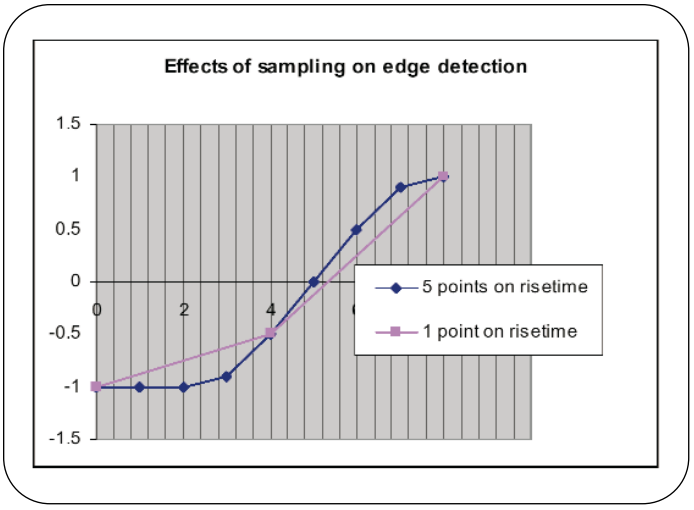
To achieve this, we want to sample the data with enough resolution to locate the edges, but not so much that we begin to limit the number of edges that we can reasonably capture. There are two tradeoffs:

- Record length versus the time it takes to process the data
- The number of edges we can capture in a given record length versus the accuracy of finding the edges

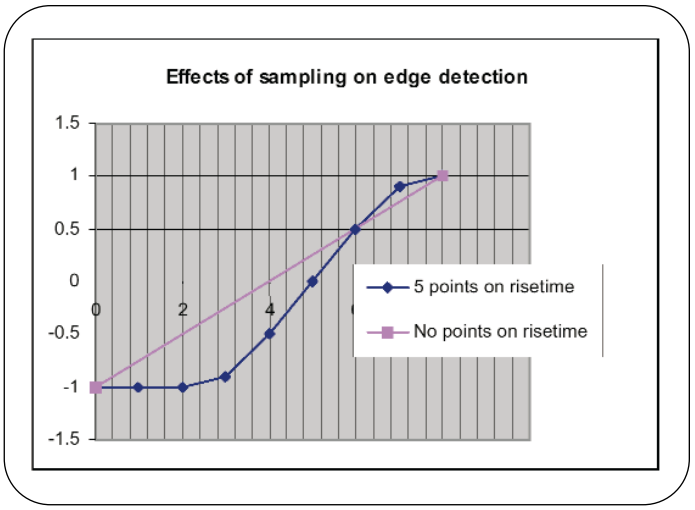
Let's explore this last tradeoff. The following graph shows an edge sampled with both three and five points on the rise time. Both pass through 0 at the horizontal value of 5. So we'll say that the edge occurs at 5.



The next graph shows the situation with five points versus a single point on the rise time. In this case, the line with a single point on the rise time passes through 0 at the horizontal value of 5.3. The data is undersampled and this undersampling has led to an error of 0.3. The result will look like jitter, but it's just an error in our measurement.




The final graph shows the situation with no points on the rise time. In this case, the line with no points on the rise time passes through 0 at a horizontal value of 4. The data is severely undersampled and this problem has led to an error of –1. Again, the result will look like jitter, but it's just an error introduced by undersampling our data in the measurement.




From the above, we can see that a sample interval (horizontal resolution) that gives us between three and five samples on the waveform rise time, will allow us to accurately find the edges. Higher resolution will just limit the number of edges in whatever record length we choose.

This first step is critical in assuring that the waveform acquisition and the subsequent analysis yield accurate results, since the error induced by undersampling will be indistinguishable from the jitter present in the signal.

Having determined the sample interval (the sample rate setting to be used for the acquisition), the waveform is captured using normal oscilloscope methodology.

## The Clock Jitter Solution using MATLAB

**Here are a few points to remember about MATLAB before you begin:**

- MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This means you can solve many technical problems in much less time than it would take to write a program in a scalar non-interactive language such as C.
- Note that many of the MATLAB Command Window instruction lines end with a semicolon (;). This causes MATLAB to perform the calculation immediately without displaying any output.
- When MATLAB encounters a new variable name, it automatically creates the variable and allocates an appropriate amount of storage.

Note: Since this document is not meant to be a detailed tutorial on MATLAB itself, all subsequent explanations are summaries of the major process steps, rather than keystroke-by-keystroke procedures.

## Export the Waveform into a File Appropriate for MATLAB

To export data stored in a TDS7000 oscilloscope, select menu bar mode, then select File> Export Setup. The Export Setup dialog box appears. Select *MATLAB* as the Data destination, *channel 1* as the source, and check the box next to Include waveform scale factors. Two files will be created. For the purposes of this example, the file(s) will be named *jitter5k.dat* and *jitter5k.hdr*. The header (.hdr) file includes four fields: the waveform record length, sample interval, trigger location and trigger time offset.

## Create the Jitter1 Function

The next step is to create a function that calls several sub-functions to solve the clock jitter problem. Start up MATLAB, select the Path Browser and define the path to the folder containing the *jitter5k.dat* and *jitter5k.hdr* files. Then start the Editor/Debugger and enter the following instructions:

```
function rmsJitter = jitter1(file,symbolRate,threshold,hysteresis)

waveform = dlmread(strcat(file,'.dat'));

header = dlmread(strcat(file,'.hdr'));

sampleInterval = header(2);
```

The first input argument *(file)* is the name of the waveform files exported from your oscilloscope. The first two assignment statements of this function read the two exported files into the *waveform* and header arrays. Then the program assigns a value to the sample interval used to collect the data (and imported as the second element in the *header* array).

Later in this exercise, we'll call the jitter1 function with the name of the waveform file *(jitter5k)* and input values for the symbol rate, threshold and hysteresis guard band.

## Measure the Timing of the Edges

First we must define all the necessary functions to solve the problem. Because of the way MATLAB works (every filename must match the function it contains), each function will be saved under its own filename.

We want to measure the timing of the edges in the raw input waveform. To do this, we'll create a function in a separate ".m" file, then call that function from the *jitter1* function.

## Create the measureEdgeTiming Function

We'll define a function called measureEdgeTiming that accepts four input arguments and returns one output argument (time). To create this function, start a new file using the File > New > M-file command in the Editor/Debugger. Type the following function definition:

*function time = measureEdgeTiming(waveform,threshold, hysteresis,sampleInterval)*

A key to this algorithm is to ensure that we are out of the guard band (hysteresis) before looking for an edge. The variable *thresholdTest* performs this task. *thresholdTest* is a state variable with a range of "cases" from 0 to 4.

Starting a FOR loop that increments through the entire sampled waveform, we will begin by finding the first voltage value that is above or below the guard band.

Case 0 deals with the initial condition:

- If the value of the sample is greater than the threshold plus the hysteresis, *thresholdTest* is set to 2 to look for a negative crossing.
- If the value of the sample is less than the threshold minus the hysteresis, *thresholdTest* is set to 1 to look for a positive crossing.
- If neither condition is met, *thresholdTest* remains zero and we continue to look for a sample outside the guard band.

Case 1 tests for a sample that is greater than or equal to the threshold—a positive edge. If the condition is met, we interpolate between two adjacent samples to locate the edge. Then a test is performed to see if the second sample is above the guard band:

- If so, *thresholdTest* is set to 2 to search for a negative crossing.
- If not, *thresholdTest* is set to 3 to search for a sample above the guard band before searching for a negative crossing.

Case 2 searches for a sample that is less than or equal to the threshold—a negative edge. If the condition is met, we interpolate between the two adjacent samples to locate the edge. Then a test is performed to see if the second sample is below the guard band.

- If so, *thresholdTest* is set to 1 to search for a positive crossing.
- If not, *thresholdTest* is set to 4 to search for a sample below the guard band before searching for a positive crossing.

Case 3 checks to see if we are above the guard band again before searching for a negative edge. This condition occurs when the second sample defining a positive edge is less than the guard band.

Case 4 checks to see if we are below the guard band again before searching for a positive edge. This condition occurs when the second sample defining a negative edge is greater than the guard band.

When the FOR loop execution is complete, all of the waveform's edges have been found and converted to crossing times (Figure 2).

```
function time = measureEdgeTiming(waveform,threshold,hysteresis,sampleInterval)
% This function finds the edges in a waveform.
% An edge is defined as the point where the waveform crosses the threshold
% The function includes a provision to define a guard band which can be used
% to reduce the sensitivity to noise.  The guard band is defined by the hysteresis
% values.
% The function will ensure that the waveform moves outside the guard band between
% successive edges.
% The edges are located by time.
% The first sample is considered as time zero.

edgeNbr = 1;
thresholdTest = 0;
% threshold test is a state variable, its values have the following meanings
% (remember, it takes two samples to define an edge)
%    0 = looking for a value outside the guard band
%    1 = looking for a positive edge. a positive edge occurs when we find a sample
%        at or above the threshold
%    2 = looking for a negative edge. a negative edge occurs when we find a sample
%        at or below the threshold
%    3 = looking for a sample above the guard band before we look for a negative
%        edge. it takes two samples to define an edge.  in this case the second
%        sample that defined a positive edge was in the guard band.
%    4 = looking for a sample below the guard band before we look for a positive
%        edge. it takes two samples to define an edge.  in this case the second
%        sample that defined a negative edge was in the guard band.

for index = 1:length(waveform)-1; % iterate through the entire waveform
   switch thresholdTest;
      case 0 % looking for a value outside the guard band
         if waveform(index) > threshold + hysteresis;
            thresholdTest = 2;
         elseif waveform(index) < threshold - hysteresis;
            thresholdTest = 1;
         end

      case 1 % below threshold, looking for positive threshold crossing
         if waveform(index + 1) >= threshold;
            fractional = (threshold - waveform(index))/(waveform(index + 1) - waveform(index));
            time(edgeNbr) = (index - 1 + fractional)*sampleInterval;
            edgeNbr = edgeNbr + 1;
            if waveform(index + 1) > threshold + hysteresis;
               thresholdTest = 2;
            else
               thresholdTest = 3;
            end
         end

      case 2 % above threshold looking for negative crossing
         if waveform(index + 1) <= threshold;
            fractional = (threshold - waveform(index))/(waveform(index + 1) - waveform(index));
            time(edgeNbr) = (index - 1 + fractional)*sampleInterval;
            edgeNbr = edgeNbr + 1;
            if waveform(index + 1) < threshold - hysteresis;
               thresholdTest = 1;
            else
               thresholdTest = 4;
            end
         end

      case 3 % looking for a sample above the threshold + hysteresis
         if waveform(index) > threshold + hysteresis;
            thresholdTest = 2;
         end

      case 4 % looking for a sample below the threshold - hysteresis
         if waveform(index) < threshold - hysteresis;
            thresholdTest = 1;
```

▶ ***Figure 2:*** *measureEdgeTiming function*

## Call the measureEdgeTiming Function

Now it's time to place a call to the function. Start the Editor/Debugger, open *jitter1.m* and add the following statement, creating an instruction that finds the edges in the waveform:

```
measuredTime = measureEdgeTiming(waveform,threshold,
hysteresis, sampleInterval);
```

This statement calls the measureEdgeTiming function with four arguments and assigns the returned result to the measuredTime array.

## Derive the Clocks Between Edges

There is always *n* number of clocks between each pair of edges. By subtracting the measured edge times between adjacent edges and then multiplying that value by the supplied symbol rate, we get a non-integer value of the number of clocks between edges. When we round the number to the nearest integer, we get a value that we then add to the clocks' array. The number stored in this array is the total number of clocks since the first edge. To calculate the number, create a FOR loop in the *jitter1.m* function

## Calculate the Average Measured Symbol Rate

Now we can find the best fit of the measured edges (in the measuredTime array) and the derived clocks (in the clocks' array) to a straight line. To do this, we'll use linear regression using MATLAB's "polyfit" function. The formula for a straight line is:

$$y = a + bx$$

where:

$a$ = the intercept (the point where a line will intersect the y-axis)
$b$ = the slope (the rate of change along the line)

The values of a and b can be produced by MATLAB's "polyfit" function. We will call the returned result *coef (1*) and *coef (2)*, respectively. We want a formula that will give us the derived time of an edge if we know the clock number. So we use:

$y$ = the time of an edge (to be stored in reconstructedTime)

$x$ = the clock number of an edge (in *clocks*)

Use the slope formula to calculate the average measured symbol rate[1] by adding the following lines to the jitter1.m function:

```
coef = polyfit(clocks, measuredTime, 1);

a = coef(2);

b = coef(1);

measuredAverageSymbolRate = 1/b;
```

[1] See *Numerical Recipes: The Art of Scientific Computing*, Cambridge Press, Chapter 14.2 for a full discussion of the equations used here.

## Calculate the Symbol Clock Error Rate

Given the average measured symbol rate, we can now calculate the symbol clock error rate. We arrive at this result by subtracting the supplied symbol rate from the average measured symbol rate and dividing the difference by the symbol rate. To the *jitter1.m* function, add:

```
measuredSymbolRateError = (measuredAverageSymbolRate
–symbolRate)/symbolRate;
```

## Graph the Waveform

Using the values calculated so far, MATLAB can graph the input waveform with the symbol rate error displayed. We will graph only a portion of the waveform —2000 samples out of 5000—so that we can see more detail in the graph. To the *jitter1.m* function, add the instructions that plot measuredSymbolRateError. The MATLAB plot function automatically opens a new Figure Window or uses an existing one.

## Reconstruct the Timing

The time of the first edge (reconstructedTime1) is the intercept of the best-fit line to the y-axis. To reconstruct the timing of all the edges (and store them in the reconstructedTime array), we apply the formula $y = a + bx$, using the intercept $a$ and slope $b$ that we previously calculated.

```
reconstructedTime(1) = coef(2);

for index = 1:length(clocks)

  reconstructedTime(index)=a + (b * clocks(index));

end
```

## Calculate the Jitter

Next we solve for the jitter, calculated by finding the difference in timing between the reconstructed edge timing and the measured edge timing. To calculate this field, add the following to *jitter1.m*:

```
jitter = reconstructedTime - measuredTime;
```

## Calculate the RMS Jitter

To calculate the RMS (root mean square) jitter, divide the normalized (average or mean) jitter array[2] by the square root of the length of the jitter array:

```
rmsJitter = norm(jitter)/sqrt(length(jitter));
```

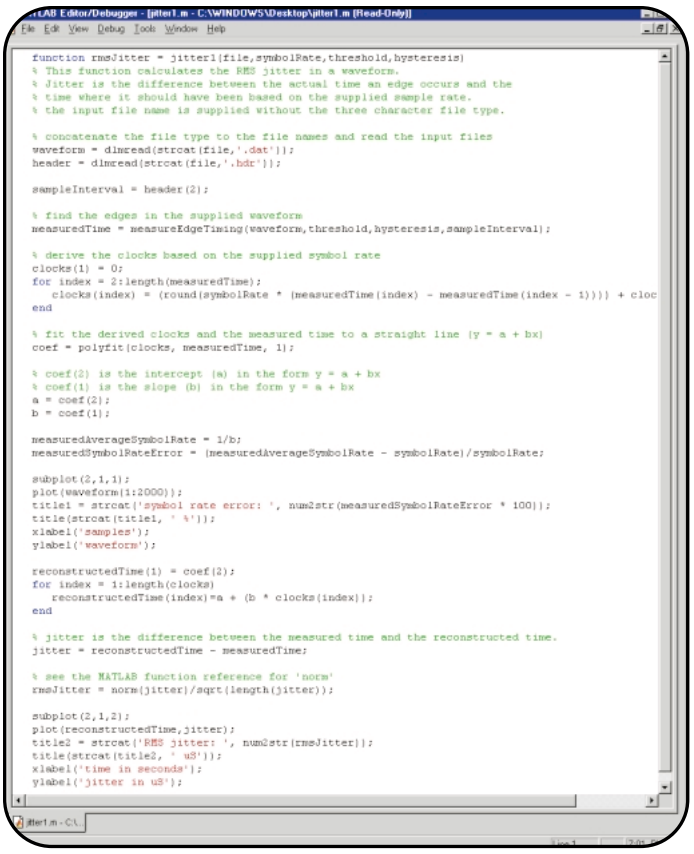
## Graph the Solution

We can instruct MATLAB to graph the results by adding the following instructions to *jitter1.m*:

```
subplot(2,1,2);

plot(reconstructedTime,jitter);

title2 = strcat('RMS jitter: ', num2str (rmsJitter));

title(strcat(title2, ' uS'));

xlabel('time in seconds');

ylabel('jitter in uS');
```

[2] The normalized jitter array is the square root of the sum of all elements in the jitter array squared.

The completed function is shown below. Be sure to save all the changes to the *jitter1* function before calling it (Figure 3).




▶ ***Figure 3:*** *The completed jitter1 function in MATLAB*

## Import the Waveform and Sample Interval, and Input Values

Now we'll import the waveform data from your oscilloscope into MATLAB. We'll call the *jitter1* function with the name of the waveform data and header files (*jitter5k.dat* and *jitter5k.hdr*) and input values for:

- The symbol rate of the clock used to generate the data
- The threshold used to determine if the waveform is a logic high (1) or low (0)
- The hysteresis that establishes a guard band around the threshold in order to reject noise in the acquired waveform

Call the jitter1 function using the following syntax:

*jitter1 ('jitter5k', 5000, 0, .1)*

MATLAB runs the *jitter1* function, assigns the returned result as the value of *rmsjitter* and, because the line doesn't end with a semicolon (;), promptly displays the answer in the Command Window.

More to the point, MATLAB also displays the two plotted graphs, as shown below (Figure 4). The upper graph is the symbol error rate, while the lower line is RMS Jitter.

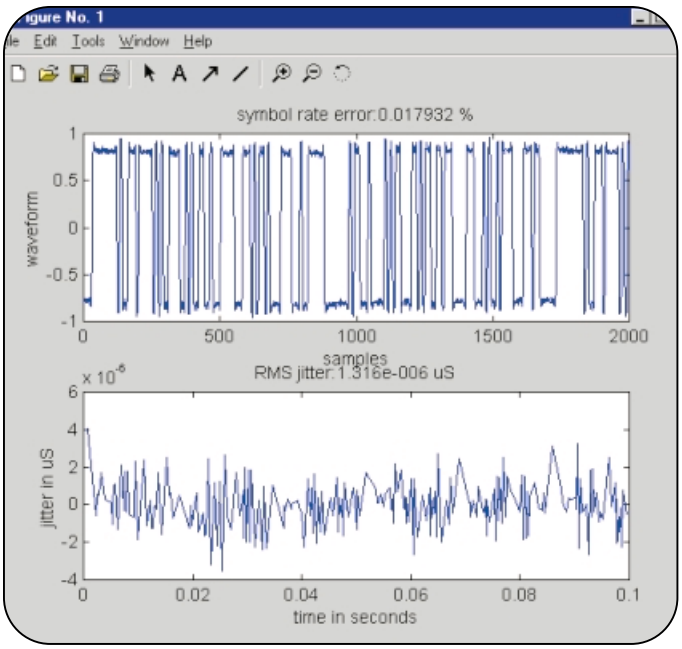



***Figure 4:*** *The waveform and jitter graphs in MATLAB.*

## Conclusion:

In this application note we have seen how the Tektronix TDS7000 Series DPO and MATLAB work together to analyze waveform jitter. With its ability to run MATLAB on the same self-contained platform as the waveform acquisition tools, the TDS7000 Series is the tool of choice for critical jitter acquisition and characterization.

▶ *TDS7054*

▶ *TDS7104*

▶ *TDS7404*

## For more information

Tektronix maintains a comprehensive, constantly expanding collection of application notes, technical briefs and other resources to help engineers working on the cutting edge of technology.

Please visit "Resources For You" on our Web site at **www.tektronix.com**

www.tektronix.com

**ASEAN Countries** (65) 356-3900

**Australia & New Zealand** 61 (2) 9888-0100

**Austria, Central Eastern Europe,**

**Greece, Turkey, Malta & Cyprus** +43 2236 8092 0

**Belgium** +32 (2) 715 89 70

**Brazil & South America** 55 (11) 3741-8360

**Canada** 1 (800) 661-5625

**Denmark** +45 (44) 850 700

**Finland** +358 (9) 4783 400

**France & North Africa** +33 1 69 86 81 81

**Germany** +49 (221) 94 77 400

**Hong Kong** (852) 2585-6688

**India** (91) 80-2275577

**Italy** +39 (02) 25086 501

**Japan (Sony/Tektronix Corporation)** 81 (3) 3448-3111

**Mexico, Central America & Caribbean** 52 (5) 666-6333

**The Netherlands** +31 23 56 95555

**Norway** +47 22 07 07 00

**People's Republic of China** 86 (10) 6235 1230

**Republic of Korea** 82 (2) 528-5299

**South Africa** (27 11) 651-5222

**Spain & Portugal** +34 91 372 6000

**Sweden** +46 8 477 65 00

**Switzerland** +41 (41) 729 36 40

**Taiwan** 886 (2) 2722-9622

**United Kingdom & Eire** +44 (0)1344 392000

**USA** 1 (800) 426-2200

For other areas, contact: Tektronix, Inc. at 1 (503) 627-6877

ISO 9001



02/01 HMH/PG 55W-14593-0



Tektronix®