Signal Computing:

Digital Signals in the Software Domain

Laboratory Manual: Matlab Edition

Fall 2015

Michael Stiber
Bilin Zhang Stiber
University of Washington Bothell
18115 Campus Way NE
Bothell, Washington 98011

Eric C. Larson
Southern Methodist University
Lyle School of Engineering
3145 Dyer Street
Dallas, TX 75205

Contents

1 The Matlab Lab, or How to Not Teach a Programming Language

Labs in this class will make liberal use of the Matlab numerical programming environment. Because this class assumes that you are an experienced computer science student, you are expected to be able to learn how to use computer tools, and how to program in new programming languages, pretty much on your own. So, the first thing you should do is check out the Matlab documentation built into the Matlab help system, or online at http://www.mathworks.com/help/matlab/index.html. Of course, you can always search online for Matlab tutorials and the like. We'll include a very brief overview of Matlab below, and then more detailed information about the code developed specifically for this class that you will be using.

1.1 Matlab in a (Very Small) Nutshell

The Matlab GUI environment is very similar to IDEs that you are already familiar with. As you might expect, there are some idiosyncrasies here and there, but nothing terribly unexpected. The major areas of difference are tools and panes that have to do with viewing variables (something in other IDEs that you'd only see when debugging a program), the command pane, and figure windows that open to display graphs. The first two of these differences have to do with the fact that Matlab is an interpreted language/programming environment. You primary area of interaction with the Matlab interpreter will be through the command window, with variable information panes providing views of variables and their contents related to your interaction. It's interesting to note that, if you want, you can run Matlab without the GUI, providing just a command line interface.

Interpreted languages have their advantages and disadvantages. One advantage is that anything you can use as a line of code in a program you can use immediately as a command on the command line. This lets you test code interactively and then copy it into the script or function you're writing. Like any IDE, Matlab includes an editor with syntax highlighting and debugger integration.

The disadvantage is the interpreted programs are slower. In Matlab, we get around this by using built-in functions that operate on entire vectors or arrays as single data objects. The core loops of the built-in functions are compiled for speed. If you make good use of those functions, Matlab code can often be as fast as completely compiled code.

Here are some things to try:

1. Immediate calculations and variables:

```
radius = 5    % Comments start with "%"
circumference = 2 * pi * radius
area = pi * radius^2
```

2. Complex numbers:

```
sqrt(-1)
x = 7 + 14j
conj(x)       % complex conjugate
abs(x)        % magnitude (e.g., for polar representation)
angle(x)      % and angle for polar rep
real(x)
imag(x)
```

3. Complex exponentials:

```
exp(j * pi)
exp(j * pi/2)
exp(j * pi/4)
```

4. Vectors:

```
v1 = [0 1 2 3]    % Four elements
v2 = [0 : 2 : 10] % like a loop (start value : increment : end value)
v3 = pi * [-0.5 -0.25 0 0.25 0.5] % All operations are vectorized
exp(j * v3)
mistake = v1 * v1     % a mistake; vector mult doesn't work this way
dotproduct = v1 * v1' % transposing will work, if you want to do this
arrayprod = v1 .* v1  % element-by-element ops include: .+, .-, .*, ./
```

5. Simple plots (note that ";" suppresses outputting results to the command window — useful if that would generate massive amounts of text, or just if you want things neat):

```
t = [0 : 0.01 : 2*pi];
x = sin(t);
plot(t, x);       % default plots points connected by lines in blue
plot(t, x, 'r');  % change the line color
plot(t, x, 'r.'); % change the plot style; zoom in to see individual points
xlabel('t, ms');  % X axis label (all of your plots should have this)
ylabel('Mag');    % Y axis label (all of your plots should have this)
title('Triangle');% Graph title (all of your plots should have this)
```

If you take a sequence of commands and save them in a file with an extension of `.m`, the result is a *script*. Assuming that the script is saved in a directory in the MATLAB search path, you can then execute the script by just typing its name (without the `.m`), just as if it were a command. If you want your code to take parameters, return a return value, and have local variables, start your code with a line like:

```
function retval = funcname(parm1, parm2)
```

Your code is now a function. MATLAB functions can take variable numbers of arguments and even return variable numbers of return values, but that's getting beyond what we need right now.

Step 1.1 It's easy to create, concatenate, extract, and modify vectors or parts of vectors. Execute the following lines of Matlab code and explain what each echoes out:

```
a = ones(1,3)
b = zeros(1,5)
x = [b, a, [1:2:12]]
x(7:end)
length(x)
x(1:2:12)
```

Also, explain the difference between the square bracket notation `[1:2:12]` and the parenthetical notation `(1:2:12)`.

Step 1.2 Consider the result of the following assignment:

```
x(7:11) = pi*(1:5)
```

Write a *single* statement that will replace the odd-indexed elements of `x` with the constant -10 (i.e., `x(1)`, `x(3)`, etc).

Step 1.3 One of the side benefits of learning Matlab is that it trains you to think in terms of parallel operations — an increasingly important skill in a profession becoming dominated by multi-core, GPU, and distributed computing. That doesn't mean you can't write loops in Matlab; it's just that your code will be more concise and efficient if you can avoid that. The efficiency arises from the fact that the vectorized Matlab commands are mostly compiled; while the loops you write are interpreted. Consider the following loop:

```
for k=0:7,
   x(k+1) = cos(k*pi/4);
end
x
```

Why is `x` indexed by `k+1` rather than `k`? What happens to the length of `x` for each iteration of the loop? Rewrite this computation without using the loop (as in list item 5). Besides the increase in efficiency from avoiding an interpreted loop, what other major efficiency results from this change?

Step 1.4 Consider the following code that plots a sinusoid:

```
t = [0 : 0.01 : 1]; % time in seconds
f = 5;              % freq in Hertz
x = sin(2*pi*f*t);
plot(t, x);
xlabel('Time (sec)');
```

Use the MATLAB editor to create a script file called `firstsin.m`, verify that you've saved it in a directory in the MATLAB path (or add that directory to the path), and test its execution by typing `firstsin` at the MATLAB command prompt. Note that you can also do:

```
type firstsin   % prints out contents of the script
which firstsin  % shows directory (useful when your code shadows built-ins)
```

If you included documentation for this script (comments at the beginning), the command `help firstsin` would also produce useful output.

Add three lines of code to your script, so that it will plot a cosine on top of the sine in a different color. Use the `hold` function to add a plot of

```
0.75*cos(2*pi*f*t)
```

to the plot. Save the plot using the MATLAB `print` command as a PNG file named `step14.png` by typing:

```
print -dpng step14
```

You should include all plots and code snippets in your lab report, following the instructions in the report rubric.

Step 1.5 You can also use Matlab to generate sounds. A pure tone is merely a sinusoid, which you already know how to generate. Let's generate one with a frequency of 3 kHz and a duration of 1 second:

```
T = 1.0;
f = 3000;
fs = 8000;
t = [0 : (1/fs) : T];
x = sin(2*pi*f*t);
sound(x, fs)
```

The vector of numbers `x` are converted into a sound waveform at a certain rate, `fs`, called the *sampling rate* (we will learn a lot more about this in this class). In this case, the sampling rate was set to 8000 samples/second. What is the length of the vector `x`?

Step 1.6 Write a new function that performs the same task as the following function without using any loops. Use the idea in step 1.3 and also consult the section on the `find` function, relational operators, and vector logicals in the MATLAB documentation.

```
function B = denegify(A)
% DENEGIFY Replace negative elements of matrix with zeros
% Usage:
%    B = denegify(A)
%
[W,H] = size(A);
for i=1:W
   for j=1:H
      if A(i,j) < 0
         B(i,j) = 0;
      else
         B(i,j) = A(i,j);
      end
   end
end
```

1.2 Trigonometric Functions and Complex Mathematics in Matlab

Step 2.1 In this step, you are asked to complete a Matlab function to synthesize a waveform in the form of:

$$x(t) = \sum_{k=1}^{N} a_i \cos(2\pi f t + \phi_k)$$

This is a sum of cosines, all at the same frequency but with different phases and amplitudes. Use the following function prototype to start you off:

```
function x = sumcos(f, phi, a, fs, dur)
% SUMCOS Synthesize a sum of cosine waves
% Usage:
%    x = sumcos(f, phi, a, fs, dur)
%        Returns sum of cosines at a single frequency f, sampling
%        rate fs, and duration dur, each with a phase phi and
%        amplitude a.
%    f = frequency (scalar)
%    phi = vector of phases
%    a = vector of amplitudes
%    fs = the sampling rate in Hz (scalar)
%    dur = total time duration of signal (scalar)
```

Include your code in your writeup. Additionally, include a plot of `x = sumcos(20, [0 pi/4 pi/2 3*pi/2], 200, 0.25);` versus time.

Hint: the MATLAB `length` function is useful in determining the number of elements in a vector; the `size` function returns both dimensions of a vector or an array.

Step 2.2 Now, let's see how complex exponentials can simplify things. Re-implement your `sumcos` function using complex exponentials. Take advantage of the fact that multiplying a complex sinusoid $e^{j2\pi ft}$ by the complex amplitude $a_i e^{j\phi}$ will shift its phase and change its amplitude. Thus, you should be able to create a *single* complex sinusoid at the given frequency `f` and then multiply it by different `a * exp(j * phi)` to get multiple phase shifted cosines. Remember that we want a real value to plot; the cosine is the real part of a complex sinusoid. Include your code in your writeup and provide a plot that demonstrates that this function produces the same result as the original implementation.

Step 2.3 Generate four sinusoids with the following amplitudes and phases:

$$x_1(t) = 6\cos(2\pi(10)t - 0.5\pi) \tag{1}$$
$$x_2(t) = 3\cos(2\pi(10)t + 0.25\pi) \tag{2}$$
$$x_3(t) = 2\cos(2\pi(10)t - 0.3\pi) \tag{3}$$
$$x_4(t) = 8\cos(2\pi(10)t + 0.9\pi) \tag{4}$$

a. Make a single plot of all four signals together over a range of t that will generate approximately 3 cycles. Make sure the plot includes negative time so that the phase at $t = 0$ can be measured. In order to get a smooth plot make sure that your have at least 20 samples per period of the wave. Include your plot in your writeup.

b. Verify that the phase of all four signals is correct at $t = 0$, and also verify that each one has the correct maximum amplitude. Use `subplot(3,2,i)` to make a six-panel subplot that puts all of these plots in the same figure, with space for two additional plots at the bottom. Use the `xlabel`, `ylabel`, and `title` functions so that the reader can figure out what the plots mean; reinforce this with your report's figure caption. (You should include the final figure, with all subplots, that results from finishing all of the parts of this step.)

c. Create the sum sinusoid, $x_5(t) = x_1(t) + x_2(t) + x_3(t) + x_4(t)$. Plot $x_5(t)$ over the same range of time as used in the last plot. Include this as the lower left panel in the plot by using `subplot(3,2,5)`.

d. Now do some complex arithmetic; create the complex amplitudes corresponding to the sinusoids $x_i(t)$: $z_i = A_i e^{j\phi_i}$, $i = 1, 2, 3, 4, 5$. Include a table in your report of the z_i in polar and rectangular form, showing A_i, ϕ_i, $\mathrm{Re}\{z_i\}$, and $\mathrm{Im}\{z_i\}$.

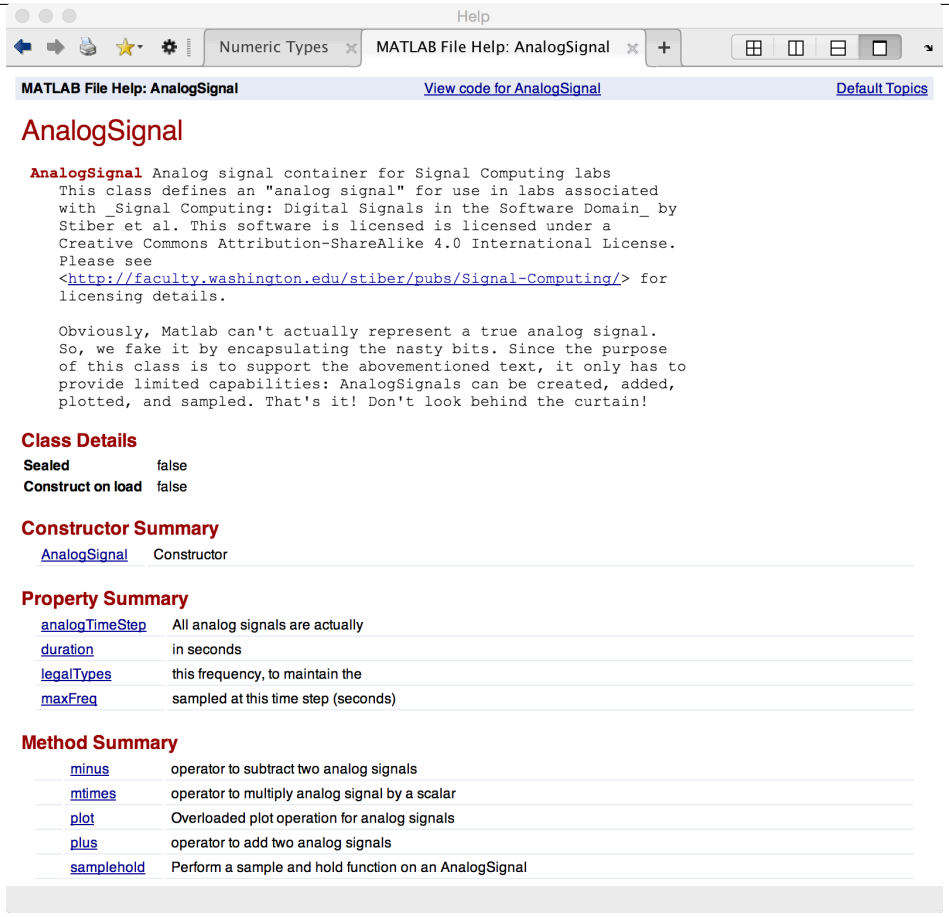

Figure 1.1: Example help screen for the `AnalogSignal` class. This example may be out-of-date; use the Matlab `doc AnalogSignal` command to get current documentation.

1.3 Representing Analog, Discrete, and Digital Signals

In our class, we will need to manipulate analog signals (real-valued signals that are functions of continuous time), discrete signals (real-valued signals that are functions of discrete time), and digital signals (discrete-valued signals that are functions of discrete time). No need to worry about the details or these; that will become clear later. The trickiest part of this is representing anything other than digital signals on a digital computer, because you can't. So, we'll need to employ two key elements of software design: information hiding and make-believe.

Information hiding is used in our implementation of analog signals. We make use of the object-oriented programming aspects of Matlab to create an `AnalogSignal` class. If you look up the documentation for `AnalogSignal` (using `doc AnalogSignal`), you'll see something like figure 1.1. The key operations on these analog signals are:

- Creating an analog signal (ex: `a = AnalogSignal('sawtooth', 2.0, 1.0, 10.0)`).

- Scaling an analog signal (ex: `b = a * 5`)
- Adding two analog signals (ex: `c = a + b`)
- Subtracting two analog signals (ex: `d = c - a`)
- Plotting an analog signal (ex: `plot(d)`)
- Sampling an analog signal (ex: `x = d.samplehold(0.1)`)

You will get a lot more experience working with analog signals shortly in this class, so for the time being just play with this a bit.

2 Let's Get Physical

In this lab, you will use Matlab to explore the effects of summing sinusoids. You will then investigate how a physical signal can be considered to be composed of a sum of sinusoids — its *Fourier series*. You will be using the Matlab `AnalogSignal` class and functions at http://faculty.washington.edu/stiber/pubs/Signal-Computing/.

2.1 Beating

In this section, you will use the Matlab `AnalogSignal` class to simulate an analog signal generator. The constructor takes the following arguments:

```
% AnalogSignal(type, amplitude, frequency, dur)
%    Where:
%    type = 'sine', 'cosine', 'square', 'sawtooth', or 'triangle'
%    amplitude = signal amplitude
%    frequency = signal frequency
%    dur = signal duration
```

Remember that, though an `AnalogSignal` is simulating an analog signal, in Matlab all functions are sampled at discrete points in time. The crufty details of this are hidden away in the class's implementation. Remember also that you can always plot an `AnalogSignal` to see what you have; include such figures in your report

Step 1.1 Verify that you can get a triangle wave. What is the code to generate and plot a triangle wave ranging from -1 to 1 Volts with a frequency of 10Hz and a duration of 1sec?

Step 1.2 Next, verify that you can generate a sine wave of 1 second duration at a frequency of 440Hz, ranging from -1 to +1. What is the Matlab code to do this? Use the `AnalogSignal soundsc` method to play this as a sound. This pitch corresponds to "standard A" on a musical scale — A above middle C. Use the Matlab `set(gca, 'XLim', [Xmin, Xmax])` function call to set the X axis limits so that the waveform is apparent (i.e., you're not just plotting a solid blob).

Step 1.3 Generate sine waves of identical range and duration, but with frequencies of 442, 444, and 448 Hz. Now, generate the sums of 440Hz and each of these new signals to generate beating akin to tuning an instrument against the 440Hz standard. Play each sum signal; can you hear the beating? Plot each sum signal for its full 1s duration. The beating "envelope" should be obvious. What is the beat frequency in each case? How does the beat frequency and amplitude relate to the textbook discussion of beating?

2.2 Fourier series representation of a physical signal

Step 2.1 Recall that any periodic signal can be represented as a sum of harmonic sinusoids. The amplitude of each harmonic is known as the Fourier Series. It may at first seem like sums of sinusoids would be poor approximations of real periodic signals, but this is not the case. We can illustrate this using a triangle wave. The formula for synthesis of a triangle wave with frequency ω_0 is a sum of harmonically related sine waves (its Fourier series):

$$x(t) = \sum_{k=0}^{\infty} \left(\underbrace{\frac{8}{\pi^2} \frac{(-1)^k}{(2k+1)^2}}_{\text{amplitude}} \underbrace{\sin((2k+1)\omega_0 t)}_{(2k+1)^{th} \text{ harmonic}} \right)$$

In this case, in the analog domain, we are dealing with frequencies in Hz, and so $\omega_0 = 2\pi f_0$. Notice that the Fourier Series of the triangle wave only uses odd harmonics (i.e., the only non-zero frequencies are $(2k+1)\omega_0 = \omega_0, 3\omega_0, 5\omega_0 \cdots$). Also notice that resulting wave will have zero mean because there is no "DC" term (i.e., $2k+1 \neq 0$ for any integer k).

Write a Matlab script that approximates a triangle wave by summing together the first 7 harmonics of its Fourier series; plot the resultant signal (i.e., use f_0, $2f_0$, $3f_0$, $\cdots 7f_0$, where f_0 =10 Hz). How does this signal compare to the triangle wave computed directly in the previous step?

Step 2.2 Another way to view a signal is in the *frequency domain*. For a signal expressed in terms of its Fourier series, the frequency representation is merely the coefficients of the harmonics. Write a Matlab function or script to compute and plot the spectrum of a triangle wave. You may find the MATLAB function `stem` useful. Note that you are *not* being asked to plot the triangle wave as a function of time; you should plot the amplitudes of the component sinusoids as a function of those sinusoids' frequencies (like the vertical lines in textbook figure 1.12). Use your code to plot the spectrum of the triangle wave from the previous step (first 7 harmonics).

3 Hello, Digital!

In this lab, you will investigate how capturing an analog signal for computer use — sampling and quantization — modifies the signal, seeing how the choices you make in the parameters for sampling and quantization affect the quality of the digitized, computer signal.

3.1 Sampling

The first step in digitization is *sample and hold*, in which the continuous analog signal is converted to a discrete-time analog signal (an analog signal that only changes its value at particular points in time). You will use the `samplehold` method to do this:

```
% samplehold Perform a sample and hold function on an AnalogSignal
% Usage:
%  x = obj.samplehold(h)
% where  obj = AnalogSignal
%        h = hold time in sec (sampling interval)
%        x = resultant sampled AnalogSignal
%
% This function produces a 1D line plot of the provided discrete or digital
% signal in a "stairstep" fashion. In other words, each value in x (which
% is a y coordinate of the plotted point (n, x)) is connected by a straight
% line to the same y value at the next sample number (n+1, x), and then by
% a vertical line to the next sample value, (n+1, x+1).
```

Step 2.1 Create an analog sine waveform ranging from -5 to 5V with a frequency of 200Hz and a duration of 2 seconds. Produce a plot with X-axis limits set to make the waveform visible (i.e., don't just make a 2s plot that tries (and fails) to show 400 cycles of the sinusoid.

Step 2.2 Use the `samplehold` method to produce sampled versions of this signal at 300Hz, 500Hz, 1000Hz, and 2000Hz. Use the Matlab `subplot` command, and the `discreteplot` function provided with this class's Matlab code, to plot the original and all four sampled signals together. Clearly, the results are not the same, and none look identical to the original sine wave. What are the two essential pieces of information about a sine wave that need to be preserved when sampling it? Does it appear that all sampled versions are equally useful in achieving this? Why or why not (in other words, your answer to this question should not be just "yes" or "no")?

Step 2.3 Let's look at aliasing in a little more detail and with a lot more numerical precision. You'll recall from the text that, once we sample a signal, we have limited the range of frequencies that we can represent in our discrete signal to the range $0 \leq \hat{\omega} \leq \pi/2$, corresponding to a range of apparent frequencies in the physical world of $0 \leq \omega' \leq \omega_s/2$

(or $0 \leq f' \leq f_s/2$). Any frequency in the original signal above $f_s/2$ will be *aliased* into the range of possible apparent frequencies. To keep things simple, we'll stick with a sinusoid; this time, make it 10Hz with an amplitude of -1 to +1 and a duration of 1s. This will make it easy to count cycles when plotted. Set up a figure that can hold three plots and plot this analog signal in the top plot.

Step 2.4 Sample this signal at 25Hz and use the `discreteplot` function to plot the sampled signal in the middle. Sample it at 15Hz and similarly plot that sampled signal at the bottom.

Step 2.5 Before examining the plots in detail, answer the following questions: For each of the two sampling frequencies, what is the range of apparent frequencies that can be represented? For each, will a sinusoid with $f = 10$Hz be aliased? If so, what will be the digital frequency and the apparent frequency of a 10Hz sinusoid?

Step 2.6 Examine the plots and count the number of up-and-down cycles in each. Don't worry that each cycle doesn't look the same, or that every other cycle seems different; just count each. You should see 10 cycles in the top graph; how many do you see in the middle and bottom? How does this compare to the theory you discussed in the previous step? If there is any discrepancy, explain it.

3.2 Analog to Digital Conversion

The last step of digitization is called "analog to digital conversion," or *quantization*. In this step, the sampled analog signal is converted to a discrete signal, with values represented by b bit integers. We will use the `quant` function provided with this class's Matlab code to perform this conversion:

```
% QUANT Quantize a sampled (discrete) signal using a prescribed
%       number of bits per point.
% Usage:
%  y = quant(x,nb,out)
% where y = digital signal quantized to 2^(nb) bits resolution
%       x = vertical points of sampled signal
%       nb = number of bits to use per point
%      out = 'raw' means output binary values: 0,...,2^(nb-1)
%      otherwise, set ouput value range = input value range
```

Step 3.1 Write a Matlab function that compares two signals by computing the *signal to noise ratio* (SNR) that results from changing one into the other (by quantization). Your function should do this by first computing *root mean squared* (RMS) error between the two. In this case, you will be comparing a sampled signal with a quantized version of it. To do

this, you should subtract the quantized signal from the sampled signal to produce a vector of differences (errors), then square each difference value using the MATLAB `.^` operator (to get squared errors), then take the mean of these squared values using the MATLAB `mean` function (mean squared error — a scalar value), and finally take the square root of that scalar result (root mean squared error). We can compute the RMS signal in a similar way, by squaring the signal values, getting the mean of those squared values, and then taking the square root. The final result should be a single value — the signal-to-noise ratio (SNR) in *decibels*. What is in the numerator and what is in the denominator of this ratio? Note that this is different than what we did in the textbook, because we are now doing the computation for a *specific* signal, not just figuring SNR for a possible *range* of signal values.

Step 3.2 Use your code to compute the SNR for a quantized sinusoid. Generate an analog signal with with a range of 0 to 5, frequency of 10Hz, and duration 2sec. Sample it at 25Hz. Use 2, 4, 8, 12, and 16 bits quantization, and plot SNR on the Y-axis versus number of quantization bits on the X-axis.

Step 3.3 Repeat Step 3.2 using a square waveform with the same parameters.

Step 3.4 Repeat Step 3.2 using a triangle waveform with the same parameters.

Step 3.5 As you double the number of bits used in quantization, how does the SNR change? How does this compare to what your learned from the textbook? Refer to specific features of your plots from Steps 3.2–3.4 to justify your answer.