Signal Computing:

Digital Signals in the Software Domain

Laboratory Manual

Fall 2014

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 Working with J-DSP

Welcome to the first Signal Computing lab! In this lab, I will introduce the J-DSP system that we will be using for the bulk of our work in this class. Note that this is intended to be an *orientation* to J-DSP. There's no way to do this without using some digital signal processing vocabulary that you may not be familiar with. However, you should not need to have a complete understanding of these concepts to complete this lab.

Note that these labs are divided into steps, with each step expecting a response and numbered "X.Y". Step X.(Y+1) is based on step X.Y; in other words, the instructions in step X.(Y+1) assume you are starting with the J-DSP setup at the end of step X.Y; it modifies that setup. On the other hand, step (X+1).1 starts with a completely new (blank) J-DSP workspace.

1.1 General Information on J-DSP

All functions in J-DSP appear as graphical blocks that are divided into groups according to their functionality. Selecting and establishing individual blocks is done using a drag-and-drop-process. Each block is linked to a signal processing function. Figure 1.1 shows the J-DSP editor environment and Figure 1.2 shows details on the drop-down menus and the signal processing functions of J-DSP. A simulation can be started by connecting appropriate blocks from left to right. Signals at any point of a simulation can be analyzed and plotted through the use of appropriate functions. Parameters in the blocks can be edited through dialog windows. Blocks can be manipulated easily (edit, move, delete and connect) using the mouse. Execution is dynamic and therefore any change at any point of a simulated system will automatically take effect in all related blocks. Select windows can be left open to enable viewing results at more than one point in the editor.

1.2 Working with J-DSP

Please note that the following notation has been used throughout the laboratory assignments:

- Block names: bold and italic, e.g., ***Plot***.
- Drop down menu item name: bold, e.g., **Basic Blocks**.
- Button: brackets, e.g., [update].
- Option to be chosen by user in a dialog box: quotes, e.g., "Gain".

The easiest way to explain some of the functions of J-DSP is to work through a simple example. To start J-DSP, go to http://depts.washington.edu/biocomp/J-DSP/, click on "Start J-DSP Locally", and press "Proceed" in the subsequent dialog window. This will load the applet; if you don't see an applet with a "Start!" button, then there is a problem with Java on your computer. Press the "Start!" button to open the J-DSP editor window.

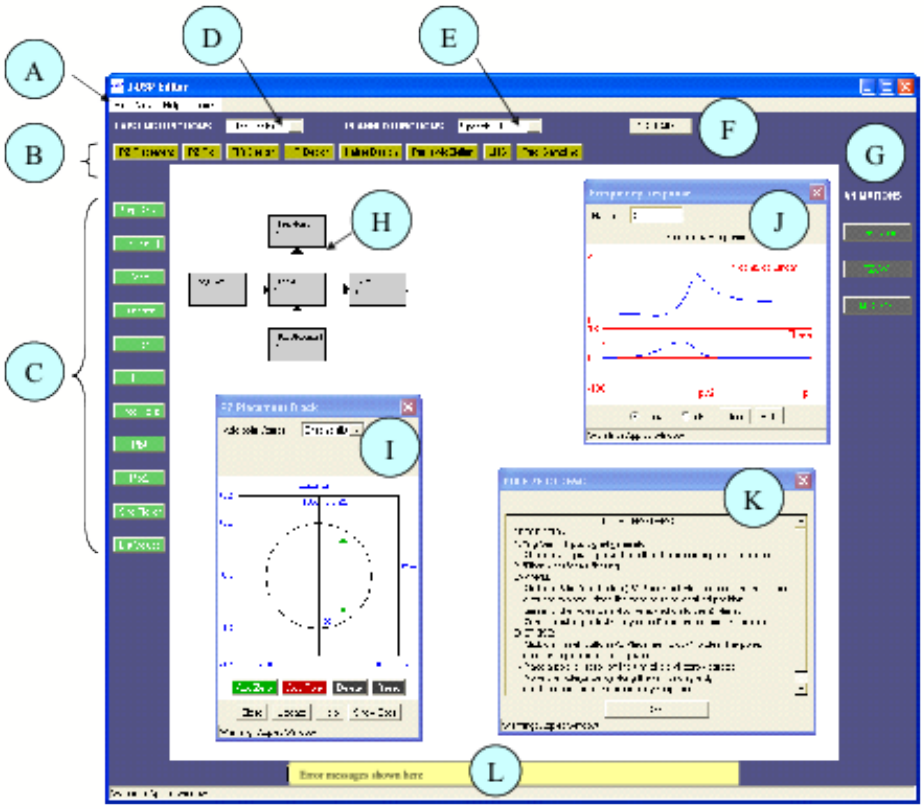


Figure 1.1: The J-DSP simulation environment. In the figure, A: Menu items; B: Filter blocks (this section changes according to the selection of D or E); C: Permanent blocks; D and E: List menu to select the group of functions; F: Disclaimer; G: Interactive visual demonstrations; H: Simulation flowgram; I: Dialog window (corresponding to the ***PZ Placement*** block in the block diagram H); J: Plot window to view the results; K: Help window provided for all the blocks; L: A field that shows error messages and warnings.

When the start button is pressed and you attempt to establish your first block a relatively large Java applet will start downloading to the browser. Therefore it may take 30 seconds or more to establish the first block for a telephone-based (28.8) internet connection but once the first block is established the program should run reasonably fast. Adjust the size of the J-DSP editor window so that you are able to view the entire work area.

Press the ***SigGen*** button on the left part of the window. Move the mouse to the center of the window and click the left mouse button. Note that you have created the signal generator block. There are two signal generators, ***SigGen*** for processing a single frame of the signal and ***SigGen(L)*** for frame-by-frame processing that is typically used in speech processing simulations. Similarly, create ***Filter*** and ***Plot*** blocks (their buttons are also on the left) as in Figure 1.3. Note that blocks cannot be placed on top of one another. There are two plot blocks, i.e., ***Plot*** (single plot) and ***Plot2*** (two plots). For now, use ***Plot***.

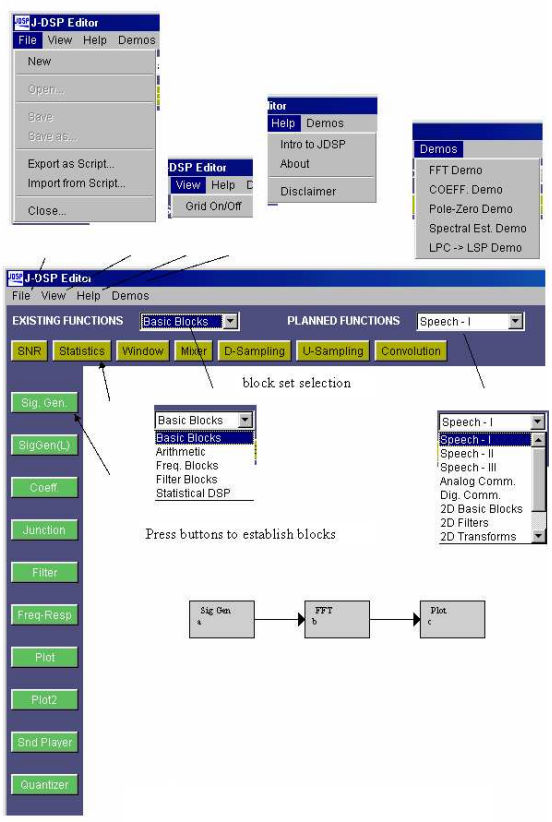


Figure 1.2: Signal processing functions and menus in J-DSP.

Note that each block has signal input(s), designated by the small triangular nodes, on the left and signal output(s) to the right. Some blocks carry parameter inputs and outputs at the bottom and top of the block respectively. For example, the ***Filter*** block has a coefficient input on the bottom and a coefficient output on the top. To select a block, click once to highlight it. You can then move it by clicking and dragging it to a new location. Try this now with your three blocks. To delete a block, simply select it and press the "delete" key on your keyboard. To link blocks, click once inside the small triangle on the right side of one block and drag the cursor to a triangle on the left side of another block. Release the mouse button to create a connection between the two boxes. Do this now to connect the output of the signal generator box to the input of the filter box. Always make the connections in the direction of the signal flow.

The ***Coeff.*** block is used to specify filter coefficients. The block is connected to the ***Filter*** block parameter input as shown in the figure; do so. Now, connect the ***Filter*** block to the ***Plot*** block and add and connect a ***Freq-Resp*** block (buttons for both are on the left) so that your editor window looks like the block diagram. Note that you can view the dialog box of each block by double-clicking on the block, as shown in Figure 1.4.

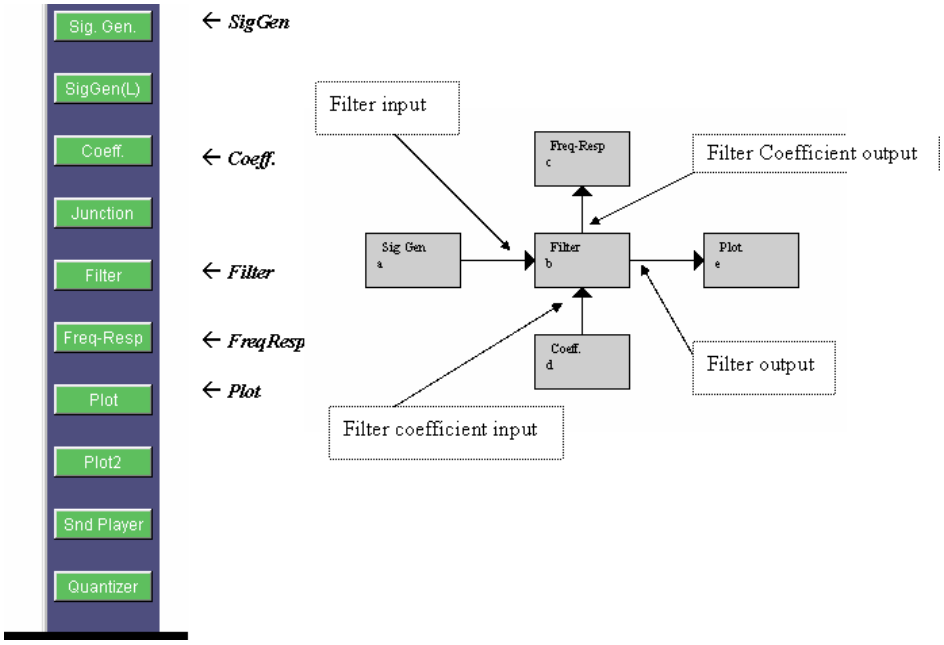


Figure 1.3: J-DSP buttons for a source-filter simulation.

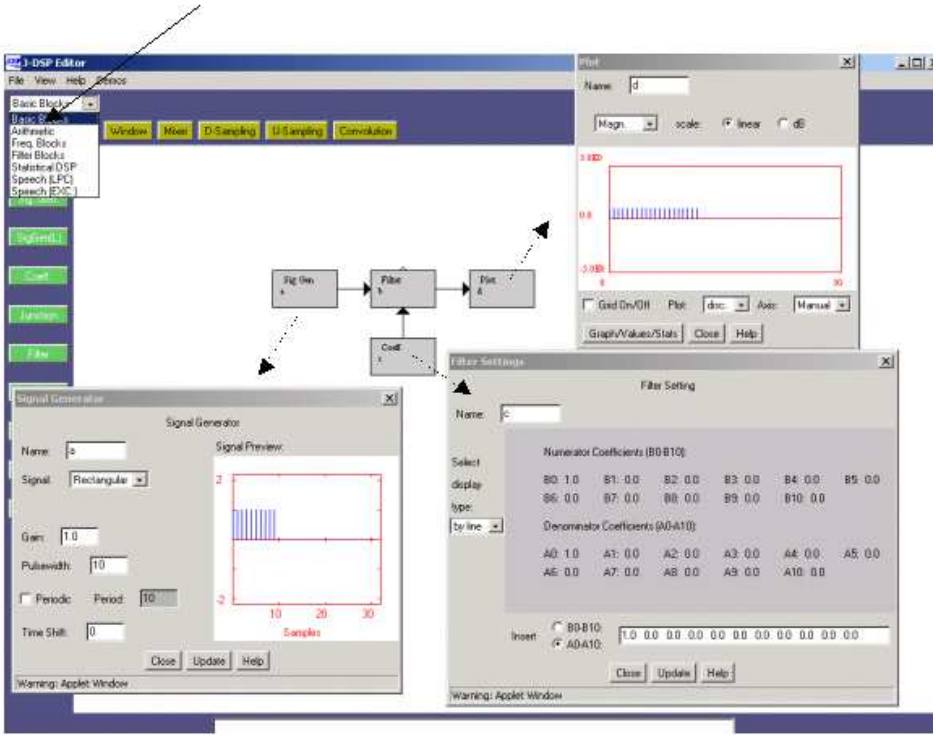


Figure 1.4: Dialog windows in J-DSP.

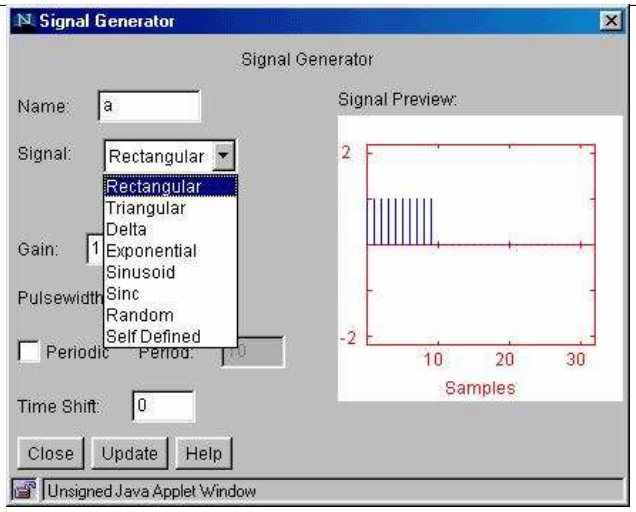


Figure 1.5: Signal generator dialog.

1.3 Choosing Signals

Let us now form a signal using the signal generator. Double click the ***Sig Gen*** box and a dialog window will appear, Figure 1.5. If you do not see a dialog window, your browser may be too old a version to work with J-DSP.

On the right side of the signal generator window, you can see a preview of part of the signal. You may change the "name" of the signal, the "gain" (the signal amplitude is ±gain), the "pulse width" (the number of samples in the entire signal), the "period" and the "time shift" by typing the desired value into the appropriate box. The signal type can be changed by clicking on the pull-down menu and selecting a signal. If you select a User-defined signal, an [Edit signal] button will appear allowing you to edit the signal. With all signals except audio, J-DSP assumes a normalized sampling frequency of 1Hz. Hence the sampling frequency in terms of radians is 2π. (Don't worry about these details right now; this will become clear after we cover "Signals in the Computer," chapter 2 of the course notes.) All frequencies are entered as a function of π, e.g., 0.1π, 0.356π, etc. Any sinusoidal frequency at or above π will result in *aliasing* (again, this will be explained in chapter 2 of the course notes).

Step 1.1 Create a sinusoid with "frequency" 0.1π, "gain" 3.75, "pulsewidth" 40. When all of the parameters have been entered, press the [update] button to update the signal preview. Remember that whenever changes are made to this box, the [update] button must be pressed in order for the changes to take effect. On the right, you can see a preview of the input signal. Note that the vertical range (or "peak-to-peak amplitude") of the signal is twice the "gain", or ±gain. In these laboratories, unless otherwise indicated, we will use "amplitude" as a synonym for the signal generator "gain" and so, for a ±3.75 signal, we will say it has an "amplitude" of 3.75. Include an image of your block diagram in your lab writeup. Count the

number of samples (vertical bars) within a period of the sinusoid. Don't forget that there are zero-height bars where the amplitude of the sinusoid is zero. How many do you have?

Step 1.2 Create a sinusoid with "frequency" π, "gain" 3.75, "pulse width" 40 (remember to press update for changes to take place). What happens?

Step 1.3 Create a sinusoid with "frequency" 1.3π, "gain" 3.75, "pulse width" 40. What happens?

1.4 Viewing Signals

Step 2 Next, we want to take a look at the ***Filter*** output in the time and frequency domain. Set the values in ***Sig Gen*** as per step 1.1. Double-click the ***Plot*** block and a new dialog window will appear. You should again see the input signal because the filter is just letting the signal pass through unaffected, since no coefficients have been set. If you move the mouse cursor within the plot, cross hairs will appear and the (x, y) coordinate at the cross hair "target" will be displayed. If you press the [Graphs/Values/Stats] button, a table with the values of the signal pops up. In the first column you see the indices of the samples and the second column shows you the values. Press the [Graphs/Values/Stats] button again to see a summary of some of the signal statistics. A third press will return us to the graph.

Step 2.1 Let us now see the filter in action. Keep the ***Plot*** window open to observe any changes. Double click the ***Coeff.*** block. You should see the display in Figure 1.6.

Step 2.2 Keep the values in ***Sig Gen*** as per step 1.1. Change the filter coefficient to $b_0 = 4$ (it will probably be easier to set the "display type" to "tabular" for setting the coefficients) and press [update]. Include an image of your block diagram and open block parameter and display windows in your writeup. Look in the plot window. You should see that the amplitude of the sinusoid has changed. What is it now?

Step 2.3 Implement a pure delay by setting $b_5 = 1$ and rest of the coefficients (including b_0) to zero and press [update]. Remember to verify after you press [update] that you correctly set the desired coefficients. Include an image of your block diagram and open block parameter and display windows in your writeup. What happens to the sinusoid?

Step 2.4 Implement a simple "low pass filter" by setting $b_0 = 0.2$ and $a_1 = -0.8$ (reset b_5 to zero) and pressing [update]. Generate a sinusoid with "gain" 1, "frequency" 0.1π, "pulse width" 256. Include an image of your block diagram and open block parameter and display windows in your writeup. What do you observe? What kind of signal do you get at the output? What is the peak-to-peak value?

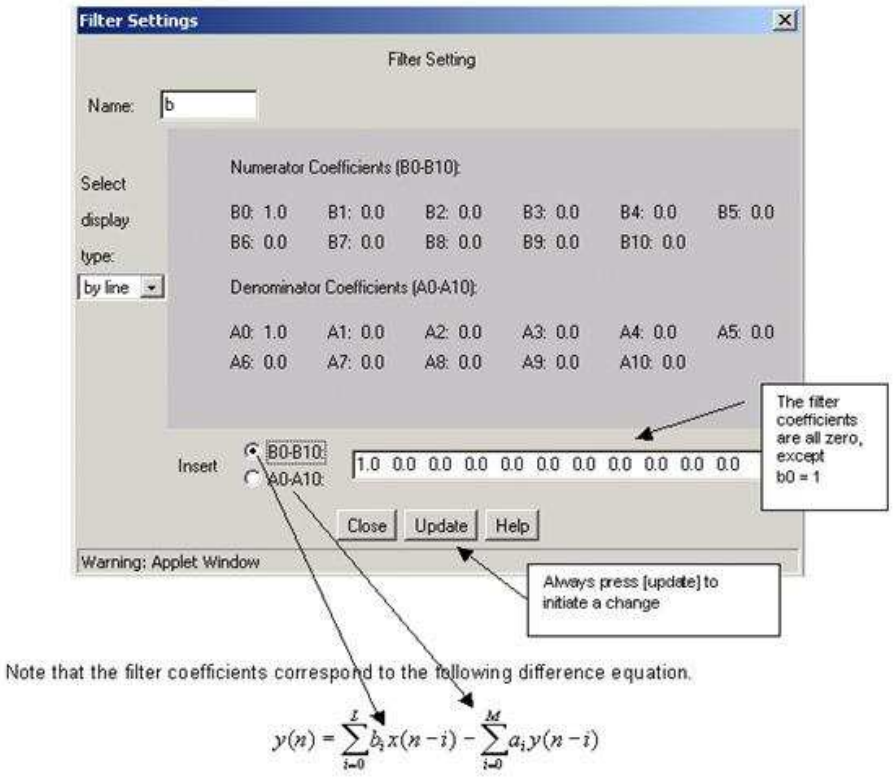


Figure 1.6: Coefficient entry in J-DSP.

Step 2.5 Select the ***Freq-Resp*** block from the panel of general blocks on the left of the window and place it to the north of the ***Filter*** block. Connect the parameter output to the ***Freq-Resp*** block. Double click the ***Freq-Resp*** block. You should see the magnitude and phase response of the filter. Change the coefficient to $a_1 = 0.8$ instead of $a_1 = -0.8$. Include an image of your block diagram and open block parameter and display windows in your writeup. What do you see in the frequency response (the magnitude plot) and output?

Step 3 To view the signal in the frequency domain, insert an ***FFT*** block between the ***Filter*** and the ***Plot*** box as shown in Figure 1.7. The ***FFT*** box can be found in the column of buttons on the left or under "Freq. Blocks" in the **Existing Functions** menu.

Step 3.1 Set the ***Filter*** parameters and input as per step 2.4. Double click on the ***FFT*** block and change the "FFT size" to 256 points and then press [Update] and [Close]. Now, you can see the amplitude of the signal in the frequency domain. The magnitude has a sharp peak approximately at 0.31, the frequency of our sinusoidal signal (0.1×3.1459). Include an image of your block diagram and open block parameter and display windows in your writeup.

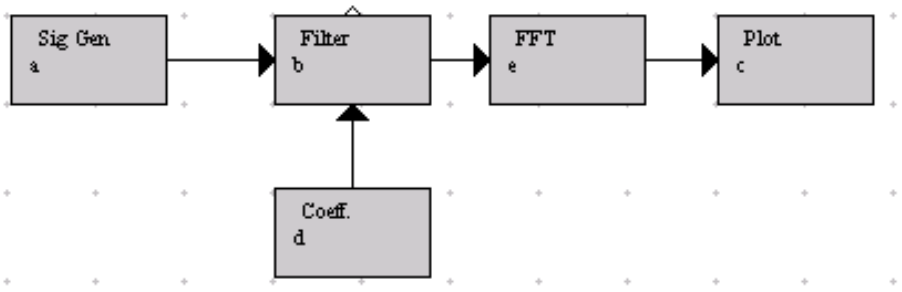


Figure 1.7: Source-filter simulation with FFT at the output.

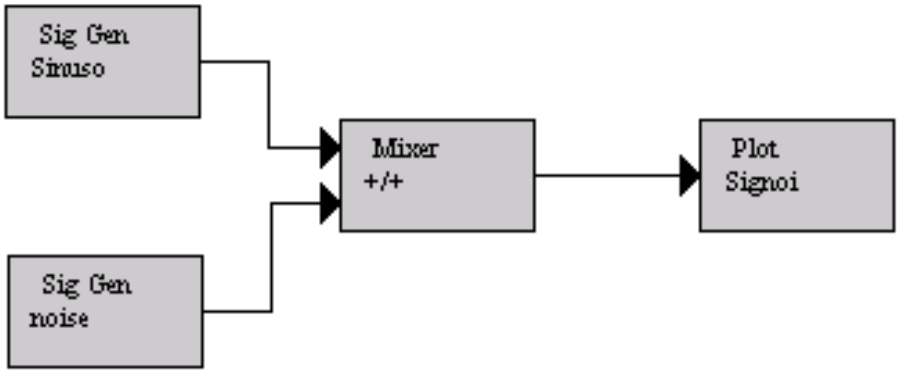


Figure 1.8: Sinewave plus noise simulation.

Step 3.2 Change the sinusoidal frequencies *only* as per steps 1.2 and 1.3 but with "pulse width" 256. Include an image of your block diagram and open block parameter and display windows in your writeup. What do you observe?

Step 3.3 Delete the ***Filter*** and ***Coef***. Set the sinusoidal "frequency" in ***Sig Gen*** as per step 1.1 but with "pulse width" 256. Now create a second ***Sig Gen*** block and a "Mixer" block (use the [Adder] button under **Basic Blocks**). Your editor window should then look like Figure 1.8.

Change the name of the first ***Sig Gen*** block to 'Sinusoi', the second ***Sig Gen*** block to 'noise' and the ***Plot*** block to 'SigNoi'. The names are restricted to six characters. Following that, we edit the ***Sig Gen*** block called 'noise'. Open the dialog window and change the "signal type" to "random". Choose a "variance" of 4 and extend the "pulse width" to 256 samples, in order to have noise over the full length of the signal. Now take a look at the output signal. In the time domain it is very hard to see that a sinusoid is present. However, if you view the signal in the frequency domain with an FFT size of 256, then you still find

a peak at approximately 0.31.

Step 3.4 Change the amplitude of the sinusoid up or down and observe the spectra (FFT plot). Try different values to make the sinusoid dominate the noise signal or be masked by the noise signal. Remember the movie "The Hunt for Red October" which was about a stealth Soviet submarine defecting? In that movie they showed sonar operators viewing FFT spectra and listening to sonar signals as they were searching for submarine propeller signatures (quasi-periodic signals) in ocean noise (random broadband signals). Stealth submarines have, among other things, weak broadband propeller signatures that can be masked easily by ocean noise.

1.5 Submitting Your Work

Please submit a lab writeup as requested by your instructor. Think of this as being a report documenting your actions, observations, and conclusions from your lab work. Thus, in responding to a question like, "what do you observe?", you might comment on the major features of a plot, such as the Y values of typical points, the range of Y values overall, the number of points in the graph, overall shape, comparison of extreme values' X and Y coordinates to other values', etc. If you're asked something like, "what can you conclude?", you would make the above observations and then use that information to produce a conclusion such as, "When I set the filter coefficient, the noise peak is reduced by a factor of 10".

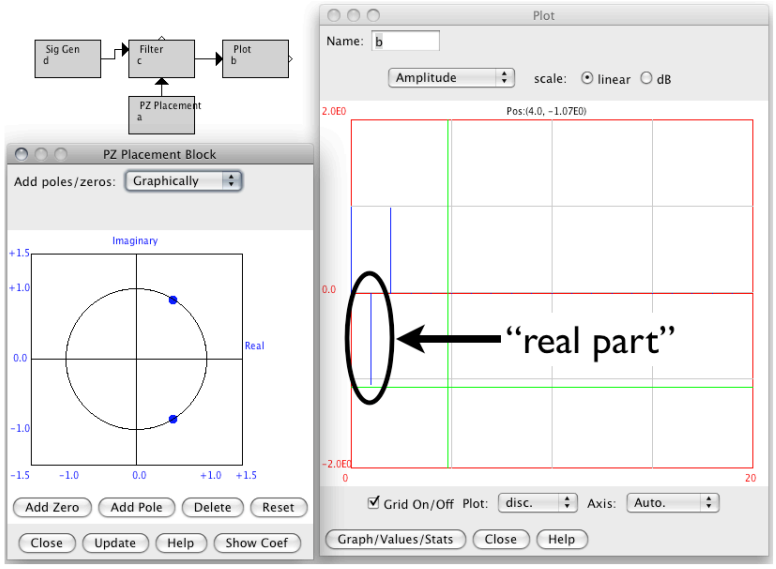


Figure 2.1: J-DSP setup for simulating a two vectors that add to be a real output, represented as sample 2 in the plot.

2 Complex Exponentials in J-DSP

In this lab you will be investigating how complex exponentials relate to sine and cosine waves. Once completed you should have a feel for how the complex plane can be used to represent *real, oscillating functions* (like sines and cosines). Moreover, you should have an appreciation for why it is easier to mathematically manipulate and analyze oscillating functions using complex arithmetic. We will be using J-DSP exclusively to illustrate key concepts of how complex numbers can represent frequency.

2.1 Complex Vectors and Exponentials in J-DSP

In this section you will use J-DSP to generate vectors and phasors in the complex plane. Specifically, you will relate changes in the complex plane to changes in the time domain.

Step 1.1 WARNING: The next step will overwrite any work not saved in J-DSP. Use the given script to generate the diagram in Figure 2.1. Download: http://faculty.washington.edu/stiber/pubs/Signal-Computing/jdsp-scripts/polezero_as_vector.txt.

Use the **File →Import from Script...** command in the J-DSP window and copy the text over. You should see a block diagram similar to that in Figure 2.1. If you do not, there was a failure loading the script, exit out of J-DSP, and try again. This script provides an interactive way to *view vectors* in the complex plane and what they look like in the time

domain. We are not yet ready to talk about every block in the J-DSP diagram. However, two blocks are of interest to us now. We will interact with ***PZ-Placement*** (block a) and ***Plot*** (block b). Open the dialog for the ***Plot*** block in the diagram if it is not already active. You should see three discrete points in the output plot (Figure 2.1).

Now open the dialog for ***PZ-Placement*** if it is not already active. You should see a plot of the complex plane with two ∘ marks on it and a large black circle that represents the unit circle. This ***PZ-Placement*** block is normally used for filter design, but we will be using it to represent a vector in the complex plane, $Re^{j\phi}$. If you drag the ∘ marks in ***PZ-Placement***, you can change the amplitude and phase of the vector and view the "real part" of the output in the time domain plot. When you do this, you will also see the three points in the output plot change in magnitude. We will be interested in what happens to the middle value in the plot (marked "real part" in Figure 2.1). You can (1) click and drag or (2) you can enter the real and complex values manually using the [Graphically] and [Manually] selections from the pop-up menu. Note that the ***PZ-placement*** block is doing more than just helping to plot the "real part" of the complex vector. For now, just know that the second point plotted in the ***Plot*** dialog corresponds to the "real part" of the complex vector. It is directly affected by the magnitude and phase of the vector. For the following questions, recall that a phasor is just a spinning vector in the complex plane.

1. If you repeatedly drag the ∘ marks in a circular motion along the black circle in ***PZ-placement***, what happens to the "real part" in the plot dialog? If you were to keep the vector magnitude constant and graph the "real part" of the plot over time as it moves around the complex plane, would you expect to see a sinusoidal wave? How would the amplitude and phase of the sinusoidal wave be affected by the complex vector?

Step 1.2 WARNING: The next step will overwrite any work not saved in J-DSP. Use the given script to generate the diagram in Figure 2.2. Download: `http://faculty.washington.edu/stiber/pubs/Signal-Computing/jdsp-scripts/polezero_as_phasor.txt`.

Use the **File →Import from Script...** command in the J-DSP window and copy the text over. You should see a block diagram similar to that in Figure 2.2. If you do not, there was a failure loading the script, exit out of J-DSP, and try again.

This script provides an interactive way to manipulate the *frequency* of phasors in the complex plane and view the sinusoidal output. We are not yet ready to talk about every block in the J-DSP diagram. Like before, two blocks are of interest to us now. We will interact with ***PZ-Placement*** (block a) and ***Plot*** (block b). Open the dialog for the ***Plot*** block in the diagram if it is not already active. You should see a cosine wave. (Note that, in the plot in the figure, the limits of the X-axis have been changed manually to have a lower limit of 65, rather than zero. Unfortunately, the axis limits don't get restored when a script is imported, so you will see the auto axis limits, rather than the ones set as in the figure. You can set the limits in the plot to match the figure yourself.)

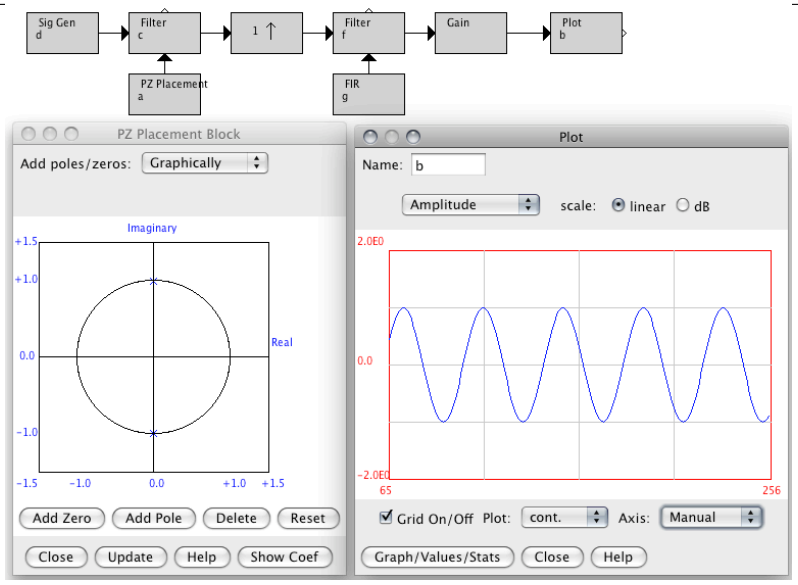


Figure 2.2: J-DSP setup for simulating two phasors that add to be a real sinusoid.

Now open the dialog for ***PZ-Placement*** if it is not already active. This block is normally used for filter design, but we will be using it to simulate a phasor in the complex plane. Recall that a phasor can be represented in polar form as $Re^{j\omega t}$ where ω represents the frequency at which the phasor rotates and R represents the magnitude. Dragging the `x` marks in ***PZ-Placement*** lets you change the ω of the resulting phasor and view the sinusoid in real time. Note that you can also change the magnitude of the complex phasors in the dialog, but this is not the same as changing the amplitude of a complex phasor. There are other things going on in this block that we are not ready to talk about - we are actually treating something called a "pole" as a rotating phasor, which is not quite right. For now, try to keep the magnitude of the phasor as close to "one" as possible (i.e., try to keep the `x` on the black line representing the unit circle).

1. Drag the phasors along the unit circle towards the (1,0) point (where the unit circle meets the real axis in the right half plane). What happens? What happens when you move away from the the (1,0) point along the unit circle? Include a screenshot of the ***Plot*** and ***PZ-placement*** dialogs in your lab report.

2. Notice that you cannot move only one of the phasors. You must move both at the same time. Does it make sense that you must move both symmetrically? Why or why not? *Hint:* think about the output as being the rotating phasors $e^{j\omega_1 t} + e^{j\omega_2 t}$, where ω_1 and ω_2 are proportional to the angles of the "poles."

2.2 Complex Exponentials

Step 2.1 Compute the following complex arithmetic by hand:

a. What is the complex conjugate of $z_1 = -1 + j0.3$?

b. Let $z_2 = 1 + j0.6$ What is $z_1 + z_2$?

c. What is $z_1 - z_2$?

d. What is $z_1 z_2$?

e. What is z_2/z_1?

Step 2.2 Compute or prove the following using complex arithmetic and Euler's Formula by hand. Recall that Euler's Formula is given by: $e^{\pm j\omega t} = \cos(\omega t) \pm j \sin(\omega t)$.

1. A complex number can be written in rectangular coordinates as $z = x + jy$. Write the relations to calculate the polar form, $z = (r, \theta)$ or $z = re^{j\theta}$.

2. Using Euler's formula, express $\cos x$ and $\sin x$ as a combination of complex exponentials.

3. Convert $\cos(\omega t + \phi)$ into the sum of complex exponentials.

4. Find expressions for 1, j, $1 + j$, $(1 + j\sqrt{3})/2$ as complex exponentials.

5. Compute $[(1 + j\sqrt{3})/2]^2$ and $(1 + j)^4$ directly (using the rectangular representations).

6. Compute $[(1 + j\sqrt{3})/2]^2$ and $(1 + j)^4$ using complex exponentials.

Step 2.3 Prove the following regarding complex conjugates:

1. Show that $(z_x z_y)^* = z_x^* z_y^*$.

2. Express $|z|^2$ as a function of z and z^*.

2.3 Complex Exponentials in J-DSP

Step 3.1 In this step, you are asked to use JDSP to create a block diagram capable of synthesize a waveform of the form: $\cos(2\pi(200)t + \phi_1) + \cos(2\pi(200)t + \phi_1) + \cos(2\pi(200)t + \phi_1) + \cos(2\pi(200)t + \phi_1)$ And plot the output of each cosine wave and their sum. Use the ***Cont. Signal*** and ***Analog Adder*** blocks in the "Analog Blocks" function set. Include a screenshot of the block diagram in your writeup. Use this diagram to complete Step 3.2.

Step 3.2 Generate four sinusoids with the following amplitudes and phases (remember to convert from rad/s to Hz and from radians to degrees):

$$x_1(t) = 5\cos(2\pi(200)t + 0.5\pi) \quad (1)$$
$$x_2(t) = 5\cos(2\pi(200)t - 0.25\pi) \quad (2)$$
$$x_3(t) = 5\cos(2\pi(200)t + 0.4\pi) \quad (3)$$
$$x_4(t) = 5\cos(2\pi(200)t - 0.9\pi) \quad (4)$$

a. Make a plots of all four signals in J-DSP.

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 the ***Analog Plot*** blocks.

c. Create the sum sinusoid, $x_5(t) = x_1(t) + x_2(t) + x_3(t) + x_4(t)$. Make a plot of $x_5(t)$ over the same range of time as used in the last plot. Include your plot in your writeup

d. Using pencil and paper: Express $x_1(t)$ through $x_4(t)$ as complex exponentials.

e. Using pencil and paper: Express $x_5(t)$ as a sum of complex exponentials.

f. Using complex exponentials, express the amplitude and phase of $x_5(t)$ (use pencil and paper with the aide of a graphing calculator, spreadsheet, or MATLAB).

3 Signals in the Computer

In this lab, you will use JDSP to explore how a physical signal can be considered to be composed of a sum of sinusoids — its *Fourier series*. You will then investigate how capturing this signal for computer use — sampling and quantization — modifies the signal. Finally, you will see how the choices you make in the parameters for sampling and quantization affect the quality of the digitized, computer signal.

3.1 Fourier series representation of a physical signal

In this section, you will use the ***Cont Sig*** block in J-DSP to simulate a analog signal generator. The ***Cont Sig*** block is capable of simulating analog signals of varying frequency content.

Step 1.1 Create a J-DSP diagram that sums together four continuous time sinusoids and plots their sum. Use the "Analog Blocks" function set and the ***Adder*** block only (See Figure 3.1). Also create a separate set of blocks that plots a continuous time triangle wave at a "frequency" of 200 Hz and "amplitude" of 1. In the next section you will simulate this triangle wave using the sum of sinusoids.

Step 1.2 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):

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

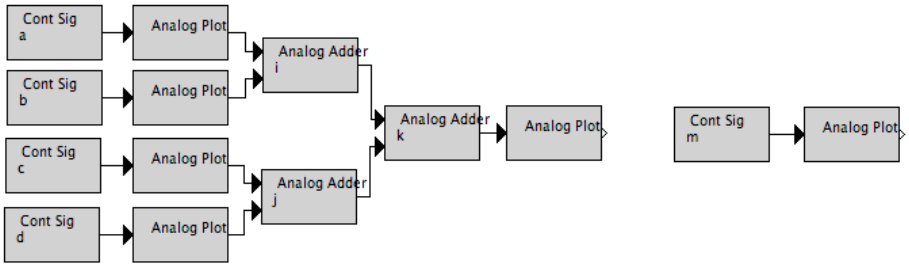


Figure 3.1: A J-DSP diagram for summing and plotting analog sine waves (left) and a single triangle wave (right).

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 be zero mean because there is no "DC" term (i.e., $2k + 1 \neq 0$ for any integer k). Use your diagram from Step 1.1 to generate and sum the first 7 harmonics of the Fourier Series of a triangle wave and plot the resultant signal (i.e., use f_0, $2f_0$, $3f_0$, $\cdots 7f_0$, where f_0 =200 Hz). How does this signal compare to the triangle wave computed directly in ***Cont. Sig***?

Step 1.3 Another way to view a signal is in the *frequency domain*. For a signal expressed in terms of its Fourier Series, the *frequency domain* representation is merely the coefficients of the harmonics. Use your favorite spreadsheet or plotting tool to compute and plot the spectrum of a triangle wave. Note that you are *not* being asked to plot the triangle wave as a function of time; you should plot the amplitudes of the Fourier Series as a function of the harmonics' frequencies (like the vertical lines in textbook figure 1.12).

3.2 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 ***Sample-Hold*** block in J-DSP to simulate this.

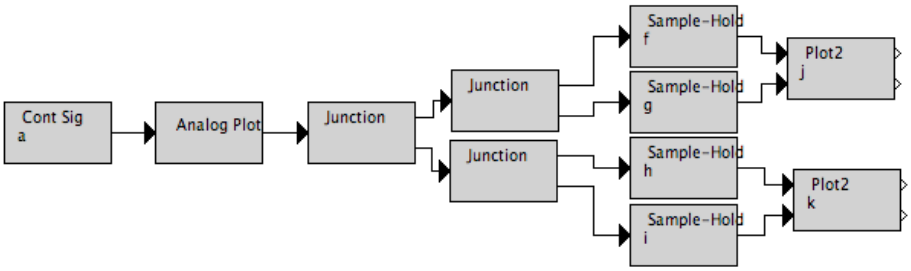


Figure 3.2: A J-DSP diagram for investigating sampling rate.

Step 2.1 Create an ***Cont Sig*** sine waveform ranging from -5 to 5V with a frequency of 200Hz (Figure 3.2, leftmost block).

Step 2.2 Use four ***Sample-Hold*** blocks to sample this signal at 300Hz, 500Hz, 1000Hz and 2000Hz. Plot the original and all four sampled signals separately (Figure 3.2). 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")?

3.3 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 ***Quantizer*** block under the "Statistical DSP" function set to perform this conversion.

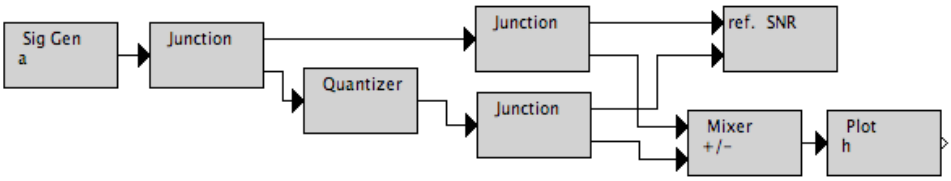


Figure 3.3: A J-DSP diagram for investigating quantization error and calculating SNR.

Step 3.1 Next we will be using the ***SNR*** block under the "Basic Blocks" function set to compute signal-to-noise ratio (SNR) as a result of quantization. Place the ***SNR*** block and open the dialog associated with it. You will see the equations used to calculate the numerator and denominator. In your own words, what quantity is calculated in the numerator and what quantity is calculated in the denominator of this ratio (Hint: can you describe the values in terms of *root mean square* (RMS) values)? Note that this SNR 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 the J-DSP diagram in Figure 3.3 to compute the SNR for a quantized sinusoid. Use the ***SigGen*** block with a "amplitude" of 5, "frequency" of 0.05π, and "Pulsewidth" of 256. Use 2, 4, 8, 12, and 16 bits quantization. Use your favorite spreadsheet or plotting software to plot SNR versus number of quantization bits (i.e., a scatter plot). Use the output of ***SigGen*** as reference for the ***SNR*** block.

Step 3.3 Plot the quantization error by using the ***Adder*** block (bottom right blocks of Figure 3.3). Set the ***Adder*** block to "subtract" by opening the dialog and switching the "+" sign to "−." You only need to include the quantization error plot for "4 bits" in your report (but be sure to view the error plots for all quantization levels).

Step 3.4 Repeat Step 3.2 using a triangle waveform with "Gain" of 5 and "Pulsewidth" of 256.

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

4 Feedforward Filters

This lab covers the basic concepts of filtering and feedforward filters. You may have also heard of feedforward filters referred to as finite impulse response (FIR) filters. In this lab, we will cover the basic idea of a filter, its mathematical representation (such as the defining equation, frequency response, and transfer function), the relationship among filter coefficients, zero placement, and filter type (low pass, high pass, band reject), and some basic properties of filters.

4.1 Overview of Filtering and J-DSP

A *digital filter* is a signal processing operation that can be described equivalently by its *defining equation*, *transfer function*, or *frequency response*. Each representation completely defines the filter. It is advantageous to use each of the different representations depending on whether you are *implementing*, *analyzing*, or *designing* an FIR filter:

$$y[n] = \underbrace{\sum_{k=0}^{M} b_k x[n-k]}_{\text{defining equation}}$$

$$Y = H(z)X$$

$$H(z) = \underbrace{\sum_{k=0}^{M} b_k z^{-k}}_{\text{transfer function}}$$

$$Y = \mathcal{H}(\hat{\omega})X$$

$$H(e^{j\hat{\omega}}) = \mathcal{H}(\hat{\omega}) = \underbrace{\sum_{k=0}^{M} b_k e^{-j\hat{\omega}k}}_{\text{frequency response}}$$

In these equations, $x[n]$ and $y[n]$ are the n^{th} samples from the input and output, respectively, while X and Y represent the entire input and output signal (all of the samples in the signal). A k-sample time delay of a signal is produced by multiplication by the delay operator, $z^{-1} = e^{-j\hat{\omega}k}$. In all three cases (but most simply for the transfer function), we can obtain insight into the filter's operation from the *coefficients*, b_k. We can do this by factoring the transfer function polynomial: its roots are the *zeros* of the filter and they can be real or complex. The placement of the zeros in the complex plane (most usefully expressed in polar coordinates) will tell us which frequencies are suppressed and to what extent those frequencies are suppressed (we can calculate each using the angle and magnitude of the zero, respectively).

J-DSP has a set of blocks that directly correspond to the concepts we have learned. These include the ***Filter*** block, which takes an input signal on the left and parameters (zero

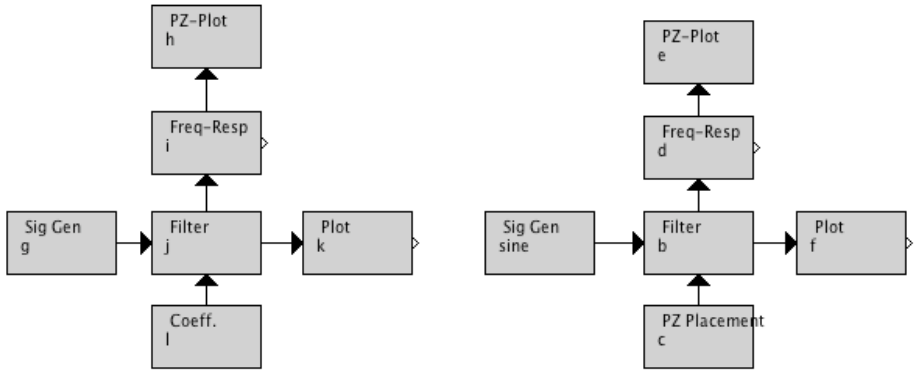


Figure 4.1: J-DSP setup for specifying filter coefficients and monitoring frequency response and zero location. (Left) Using the ***coeff*** block. (Right) Using the ***PZ-Placement*** block

locations or coefficients) at the bottom, and produces an output signal on the right and can have its characteristics (frequency response or zero locations) monitored via a connection from the top. We will use ***Filter*** in two different ways:

1. We will specify the coefficients of the transfer function or generating equation using the ***Coeff.*** block and monitor the filter's output (using ***Plot***), frequency response (using ***Freq-Resp***), and zero placement (using ***PZ-Plot***), as in Figure 4.1 (left).

2. Instead of defining the filter using the ***Coeff*** block, we will place zeros directly in the complex plane using the ***PZ Placement*** block (which has an option to show the corresponding filter coefficients), as shown in Figure 4.1 (right).

Remember, we can define a specific filter using either of the methods above. Sometimes it is easier to understand the filter using zeros and sometimes it is easier to use the coefficients directly. Either method can be used to represent the same filter.

4.1.1 From Filter Coefficients to Transfer Function and Frequency Response

Given the coefficients of an FIR filter we can solve for the zero locations and the frequency response. For example the two-point averaging system is given by:

$$y[n] = \frac{1}{2}x[n] + \frac{1}{2}x[n-1]$$

we can find the transfer function by rewriting the filter using the delay operator, z:

$$Y = \frac{1}{2}X + \frac{1}{2}z^{-1}X \tag{5}$$

$$H(z) = \frac{Y}{X} = \frac{1}{2}(1 + z^{-1}) \tag{6}$$

If we're interested in the *zero location* we can then multiply $H(z)$ by z/z to obtain:

$$H(z) = \frac{\frac{1}{2}(z+1)}{z} \tag{7}$$

The root of the numerator, $z = -1$ is the location of the only zero (the root(s) of the denominator for an FIR filter are always at $z = 0$, and do not affect the frequency response). We can also derive the frequency response from this by remembering that $H(e^{j\hat{\omega}}) = \mathcal{H}(\hat{\omega})$ or that $z = e^{j\hat{\omega}}$. From the zero location, $z = -1$, we can immediately tell that the frequency response is zero at $e^{j\hat{\omega}} = -1$ or $\hat{\omega} = \pi$. With a zero at an angle of $\hat{\omega} = \pi$, this is a low-pass filter. As a warm-up, use the J-DSP setup of Figure 4.1 (left) to verify the expected zero placement and frequency response.

4.1.2 From Zero Placement to Filter Coefficients

When we are given the zero placement, we can very easily determine the filter coefficients because those zeros are the roots of a factored polynomial. For example, given two complex conjugate zeros, z_1 and z_2 (i.e., the real parts are equal, $\mathrm{Re}[z_1] = \mathrm{Re}[z_2] = \mathrm{Re}[z_{1,2}]$, and the imaginary parts are negatives of one another $\mathrm{Im}[z_1] = -\mathrm{Im}[z_2]$ or, equivalently in polar coordinates, $z_1 = re^{j\omega_0}$ and $z_2 = re^{-j\omega_0}$), the transfer function is:

$$\begin{aligned}
H(z) &= (z - z_1)(z - z_2)/z^2 && (8)\\
&= (z^2 - (z_1 + z_2)z + z_1 z_2)/z^2 && (9)\\
&= 1 - 2\,\mathrm{Re}[z_{1,2}]z^{-1} + r^2 z^{-2} && (10)\\
&= 1 - 2\,\mathrm{Re}[r(\cos(\omega_0) \pm j\sin(\omega_0))]z^{-1} + r^2 z^{-2} && (11)\\
&= \underbrace{1}_{b_0} - \underbrace{2r\cos(\omega_0)}_{b_1} z^{-1} + \underbrace{r^2}_{b_2} z^{-2} && (12)
\end{aligned}$$

At this point, we can rewrite the transfer function as the filter's generating equation, using the delay operator z^{-k}, $y[n] = x[n] - 2r\cos(\omega_0)x[n-1] + r^2 x[n-2]$. This allows us to read off the filter coefficients: $b_0 = 1$, $b_1 = -2r\cos(\omega_0)$, and $b_2 = r^2$.

4.2 Frequency Response and Pole-Zero Plots

Step 1.1 Consider a filter that computes a running average of three points of our input signal (a *three-point averager*):

$$y[n] = \frac{1}{3}\sum_{k=0}^{2} x[n-k] = \frac{1}{3}x[n] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n-2] \tag{13}$$

a. Draw a block diagram for this filter.

b. How many zeros will this filter have?

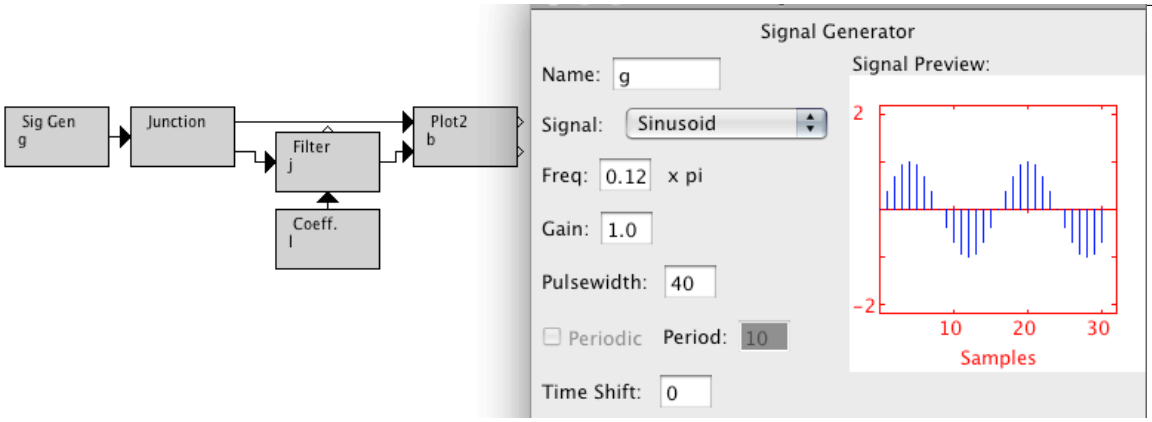


Figure 4.2: J-DSP setup for Step 1.2 (e, f, and g).

c. Find and sketch the zero locations using pencil and paper, then use J-DSP to verify this.

d. Sketch the magnitude of the frequency response as a function of $\hat{\omega}$ from the plot of pole locations and verify this using J-DSP. How does the minimum of the magnitude of the frequency response relate to the polar representation of the zero locations? What kind of filter would you say this is?

Step 1.2 A *first-difference* filter is an approximation to a discrete derivative operation. Its defining equation is:

$$y[n] = x[n] - x[n-1] \tag{14}$$

a. Draw a block diagram for this filter.

b. Derive the transfer function, $H(z)$, for this filter. From this, determine the expression for the frequency response, $\mathcal{H}(\hat{\omega}) = H(e^{j\hat{\omega}})$.

c. From the transfer function, determine the filter's zero locations and sketch them. Check your results with J-DSP.

d. From the zero plot, sketch the magnitude of the filter's frequency response as a function of $\hat{\omega}$. Use J-DSP to check your results. What kind of filter would you say this is?

e. Use J-DSP to simulate this filter's response to the following input. Set the ***Sig Gen*** to produce a sinusoid with "frequency" 0.125π, "gain" 1, "pulsewidth" 40. The diagram is shown in Figure 4.2.

f. Examine the plots of X and Y in J-DSP. Note that Y appears to be a scaled and shifted sinusoid of the same frequency as X. The exception is the first point, $y[0]$. Explain why $y[0]$ is different.

g. Estimate the frequency, amplitude, and phase of Y directly from its plot (ignoring $y[0]$).

h. To compare these measurements to theory, use your expression for the filter's frequency response to calculate the amplitude and phase at a frequency of $\hat{\omega} = \pi/8$. How do these compare to what you determined from the J-DSP plots?

Step 1.3 Just as we can compute a discrete first derivative with a first-difference filter, we can compute a discrete second derivative with a *second difference filter*.

a. Use your expression for the transfer function of the first difference filter and your knowledge that the combined transfer function of two filters cascaded, or connected in series, is the product of their individual transfer functions to determine the transfer function for a second-difference filter.

b. Draw a block diagram for this filter.

c. Determine the filter's zero locations and sketch them. Check your results using J-DSP.

d. From the zero plot, sketch the magnitude of the filter's frequency response as a function of $\hat{\omega}$. Use J-DSP to check your results. What kind of filter would you say this is?

Step 1.4 Consider a feedforward filter with complex conjugate zeros at $z_{1,2} = -0.5 \pm j0.5$.

a. Determine the filter coefficients.

b. Use J-DSP to plot the frequency response of the filter.

c. What are the effects of the zeros on the frequency response? What kind of filter would you call this?

4.3 Linearity and Cascading Filters

Step 2.1 A system is called *linear* if a sum of different inputs produces an output that is the sum of the outputs for the inputs taken individually. Perform a simple test of the linearity of this filter by doubling the input amplitude in J-DSP ($X' = 2X = X + X$). How does the new output amplitude compare to the old one?

Step 2.2 In one of the self-test exercises in the class notes, two filters with transfer functions $H_1(z) = b_0 + b_1 z^{-1}$ and $H_2(z) = b'_0 + b'_1 z^{-1}$ were connected in series, and it was shown that they could be connected in either order to produce the same composite effect (the same overall transfer function). Redo this exercise using the *defining equations* for the two filters, i.e., $y_1[n] = F_1(x[n])$ for the filter with transfer function $H_1(z)$ and $y_2[n] = F_2(x[n])$ for the filter with transfer function $H_2(z)$. In other words, show that $F_2(F_1(x[n])) = F_1(F_2(x[n]))$.

Step 2.3 Use J-DSP to implement a system in which the output of the ***Sig Gen*** is connected to the three-point averager of step 1.1, the output of which is in turn connected to the first-difference filter of step 1.2. Set the ***Sig Gen*** to output a periodic, rectangular waveform with amplitude of 1, pulse width of 10, and period of 20. This is a 50% duty cycle square wave. Plot the final output waveform. What does it look like?

Step 2.4 Change your J-DSP simulation so that the first difference filter is first and the three-point averager is second. Plot the output. How does the output of this configuration compare to that of step 3.3?

5 More Feedforward Filters

This lab discusses the different ways in which to express a discrete feedforward filter. When completed you should feel comfortable representing a filter in any of its equivalent forms and know the different advantages of expressing a filter in these forms. When you get down to it, feedforward filters can be expressed in the following equivalent ways:

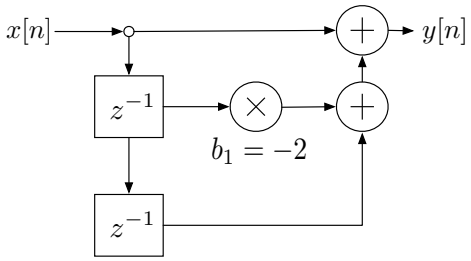
1. Defining equation, $y[n]$ as a (linear) function of $x[n], x[n-1], \ldots$, or equivalently as this equation's coefficients.
2. Transfer function, $H(z)$.
3. Frequency response (i.e., the expression for $\mathcal{H}(\hat{\omega})$).
4. Zero locations, z_i, often plotted in the complex plane.
5. Amplitude $|\mathcal{H}(\hat{\omega})| = |H(e^{j\hat{\omega}})|$ (and phase $\theta(\hat{\omega})$) of the frequency response, often plotted as functions of $\hat{\omega}$.
6. Block diagram.

In this lab, you will be given a number of filters, each expressed in one of the above ways, for each, you are asked to generate the other representations using pencil and paper. For our purposes here, it will only be necessary to sketch the amplitude of the filter's frequency response (for item 5 above). For items 4 and 5, please use J-DSP to check your answers. If there is disagreement between your answers and what you get in J-DSP, please spend some time out of class discussing the discrepancy with your classmates or with me to determine its source.

Step 1: Three-point averager Yes, we've done some of this already, so it's a good, familiar place to start. The defining equation is $y[n] = \frac{1}{3}x[n] + \frac{1}{3}x[n-1] + \frac{1}{3}x[n-2]$; determine the other representations.

Step 2: First-difference filter A *first-difference* filter is an approximation to a discrete derivative operation. Its transfer function is $H(z) = 1 - z^{-1}$. Determine the other representations.

Step 3: Second-difference filter Just as we can compute a discrete first derivative with a first-difference filter, we can compute a discrete second derivative with a *second difference filter*. Its block diagram (in a little trickier format) is:



After you've developed the alternative representations, show that this filter can also be implemented as a cascade of two first-difference filters.

Step 4: Complex conjugate zeros Feedforward filters with real coefficients in their defining equations have either real or complex conjugate zeros. Re-represent the filter with zeros $z_{1,2} = 0.5 \pm j0.5$ and $z_3 = 0.75$.

Step 5: Symmetric filters Feedforward filters with symmetric coefficients have interesting properties, such as linear phase terms. Even symmetric filters have a midpoint around which they mirror their coefficients. Re-represent the filter with a coefficients given by $\{1, 2, 2, 1\}$ (i.e., $y[n] = x[n] + 2x[n-1] + 2x[n-2] + x[n-3]$). In this filter, the coefficients {1,2} are mirrored.

6 The Z-Transform and Convolution

This lab covers the z-transform, used to convert arbitrary digital signals to the frequency domain. It also exercises the relationship between a filter's transfer function and impulse response and how the operations of multiplication and convolution, respectively, can be used to compute a filter's output.

6.1 The z-transform, Transfer Function, & Impulse Response

A discrete signal $x[n]$ has a z-transform $X(z)$ defined by the following equation:

$$X(z) = \sum_{n=0}^{\infty} x[n]z^{-n}$$

With this definition lets investigate a feed forward filter with ten coefficients, $\{b_0, b_1, \cdots, b_9\}$. Recall that the ***Filter*** and ***Coeff.*** blocks of J-DSP allow us to specify a filter in terms of its *coefficients*, but we can also define it in terms of its *transfer function*. Considering the b_k coefficients of the feed forward filter, the ***Filter*** block implements the transfer function:

$$H(z) = \sum_{k=0}^{9} b_k z^{-k} \tag{15}$$

In previous labs we have computed the transfer function using the delays of the *defining function*. Mathematically, we were actually taking the z-transform of the *impulse response*! In this example, the impulse response is:

$$h[n] = \sum_{k=0}^{9} b_k \delta[k] \tag{16}$$

where $\delta[k]$ is the unit impulse and only has value at k. $H(z)$ and $h[n]$ form a z-transform pair, $h[n] \overset{z}{\leftrightarrow} H(z)$. It should now be obvious why feedforward filters are also known as finite impulse response filters – their impulse response only has a *finite* number of values. To compute the output, $y[n]$, using the impulse response we use *convolution*. Namely, we *convolve* the input, $x[n]$, by the impulse response, $h[n]$,

$$y[n] = x[n] * h[n] = \sum_{k=0}^{9} x[k]h[n-k] \tag{17}$$

Alternatively, we can compute a filter's output by multiplying the transfer function by the z-transform of the input to yield the z-transform of the output:

$$Y(z) = H(z)X(z) \tag{18}$$

From a practical point of view, of course, it makes more sense to implement a filter in terms of its impulse response. However, for filters with long impulse responses, it is sometimes more convenient to represent them mathematically using the transfer function (which we now know is just the z-transform of the impulse response!). So, in effect, the J-DSP blocks allow us to invert the z-transform of various signals.

6.2 Z-Transforms

Step 1.1 On paper, compute the z-transform, $X(z)$, of

$$x[n] = \begin{cases} (-1)^n & n \geq 0 \\ 0 & n < 0 \end{cases} \tag{19}$$

Note that this is an infinite geometric series. What are the locations of any pole(s) (roots of the denominator polynomial) or zero(s) (roots of the numerator polynomial)?

Step 1.2 Evaluate the frequency response of $X(z)$ from step 1.1, $X(z)\big|_{z=e^{j\hat{\omega}}}$. What kind of filter is this?

Step 1.3 Consider the z-transform:

$$X(z) = 1 - 2z^{-1} + 3z^{-3} - z^{-5} \tag{20}$$

Write the inverse z-transform, $x[n]$, as a table of values for corresponding n values.

6.3 Impulse Response

Step 2.1 Consider a filter with a transfer function

$$H(z) = 1 + 5z^{-1} - 3z^{-2} + 2.5z^{-3} + 4z^{-8} \tag{21}$$

What is the defining equation for this filter, $y[n] = F(x[n])$?

Step 2.2 What is the output sequence of the filter of Step 2.1 when the input is $x[n] = \delta[n]$?

Step 2.3 The impulse response of a filter is $h[n] = x[n] + 2x[n-1] + x[n-2] - x[n-3]$, or equivalently, $h[n] = \{1, 2, 1, -1\}$, $n = \{0, 1, 2, 3\}$. Determine the response of the system to the input signal $x[n] = \{1, 2, 3, 1\}$, $n = \{0, 1, 2, 3\}$. Use J-DSP to check your results. Include an image of the J-DSP block diagram with plots of the ***Sig Gen*** signal and the filter output in your report. Note that the ***Sig Gen*** block has a *user-defined* "signal" option; use [reset] to reset all the signal values to zero before entering your own. Additionally, you can view the values in the ***Plot*** block.

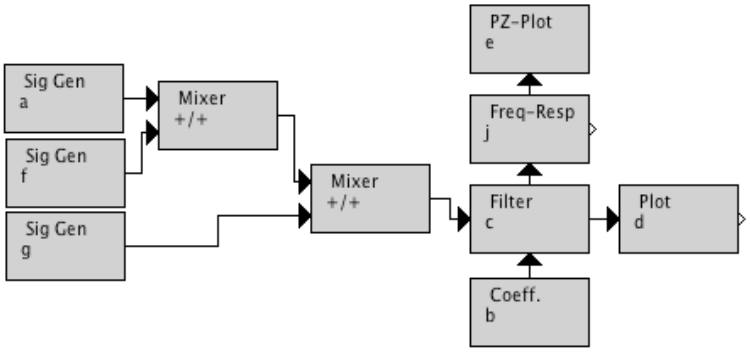


Figure 6.1: An example diagram for use in problem 2.5

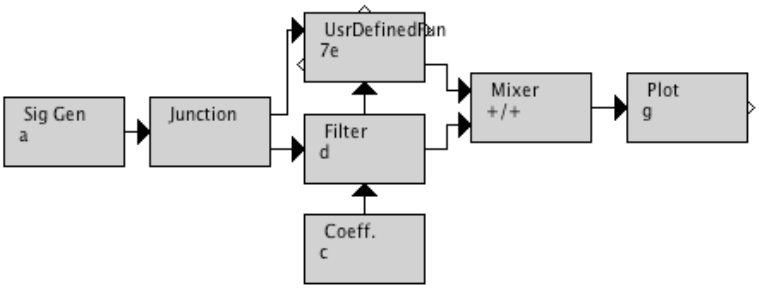


Figure 6.2: A diagram to plot the difference between ***UserDefinedFun*** function output and the ***Filter*** block output for use in step 2.6.

Step 2.4 Change the input to the filter of Step 2.3 to be $\delta[n]$, using the "signal" set to "Delta". What are the output values? How do they compare to the impulse response? Include a plot of the filter output values in your report.

Step 2.5 Use J-DSP to determine the output of the filter $\{1/3,1/3,1/3\}$, $n = \{0, 1, 2\}$ for the input:

$$x[n] = 4 + \sin[0.25\pi(n-1)] - 3\sin[(2\pi/3)n] \tag{22}$$

An example diagram can be seen in Figure 6.1. Include an image of the J-DSP block diagram with plots of the filter input and output in your report. Is the result expected? Why or why not?

Step 2.6 Create your own ***UserDefinedFun*** block to implement convolution in J-DSP. To test your function, make sure it works exactly like the ***Filter*** block in J-DSP. Use the diagram in Figure 6.2 to plot the difference between your function output and the ***Filter*** block output. Use the filter $\{1/3,1/3,1/3\}$, $n = \{0, 1, 2\}$ from step 2.5. Note: refer to section 6.5 for how to use the ***UserDefinedFun*** block in J-DSP.

6.4 Canceling Sinusoidal Components

Filters can be designed to cancel sinusoids. Construct a filter in J-DSP with the following impulse response:

$$h[n] = \delta[n] - 2\cos(\pi/4)\delta[n-1] + \delta[n-2] \tag{23}$$

Step 3.1 Plot the frequency response for this filter. What are the zero locations?

Step 3.2 Use as an input to this filter the signal $x[n] = \sin\hat{\omega}n$, using the two frequencies $\hat{\omega} = \pi/2$ and $\hat{\omega} = \pi/4$. Simulate the filter in J-DSP for each of these two inputs. Plot the filter output for each. When do we get cancellation?

Step 3.3 Can you modify the filter coefficients to cancel the other sinusoid? If so, show your work.

6.5 Example User Defined J-DSP Function

Open J-DSP and navigate to the "Advanced" function set. There is one block called ***UserDefinedFun***. Place it now and double click to open the dialog. You will see a dialog with example text for a prototype java class. In particular the class will look like:

```
public void myCode(double[]x1,double[]x2, double[]y1, double[]y2,
                   double[]b1,double[]a1, double[]b2, double[]a2,
                   double para1, double para2, double para3)
{
  //                          /\(3)
  //                  ____________________
  //            (0)<|                    |>(4)
  //                |       BLOCK        |
  //            (1)<|                    |>(5)
  //                  ____________________
  //                          \/(2)

  // x1, x2 - input at pin 0 and pin1
  // y1, y2 - output at pin 4 and pin5
  // b1 - FIR Coefficients of the filter at input pin 2,
  // a1 - IIR Coefficients of the filter at input pin 2
  // b2 - FIR Coefficients of the filter at output pin 3,
  // a2 - IIR Coefficients of the filter at output pin 3

  // Para1, Para2 and Para3 are the variables that could be used
  // in the code and can be controlled from the block Dialog.

  // Paste your code here
  // compile the file
```

```
  // upload the .class file
}
```

You can copy this code as a prototype and create your own signal processing algorithms! For now, you will mainly deal with the x and y arrays. Each of the arrays is of length 256. You can access this like a any regular array in java, *x1.length=256*. We also have access to four variable length arrays, *a1 a2 b1 b2* which contain any filter coefficients as inputs or outputs to the function.

Lets start by creating our own .java file. Copy the example code on the next page and save it as a .java file using your favorite java text editor or plain text editor. The example code creates a weighted sum of the entries in each index of *x1*. Namely, it goes through each value of *x1* and uses the *b1* filter coefficients to weight and sum the value into *y1*. Save the file as "MyFunction1.java" and then compile the file using your favorite compiler tool for java (javac from the command line works fine for linux and mac or The Java SE Development Kit 6 (JDK 6) can be used in windows). This creates "MyFunction1.class."

After compiling, remember where you saved the "MyFunction1.class" compiled file. You will need to tell J-DSP where the file is located. Go back to the J-DSP block diagram and press [open]. Navigate to the "MyFunction1.class" compiled file and press "open". That's it! J-DSP will automatically use the function when you attach inputs and outputs. To test the function we just made you can connect a ***SigGen*** block to the top left pin and a ***Coeff*** pin to the bottom as shown on page 33. You should see a weighted sum of the input when you plot the output. You should be able to change the output by changing the weighting coefficients from ***Coeff***. In the example on page 33 we are taking a weighted sum (using the weights in *b1*) of each coefficient in *x1* three times. You are now ready to start developing your own functions in J-DSP!

```
public class MyFunction1
{
  public void myCode(double[]x1,double[]x2, double[]y1, double[]y2,
      double[]b1,double[]a1, double[]b2, double[]a2,
      double para1, double para2, double para3)
  {
    //                          /\(3)
    //                  ------------------
    //          (0)<|                      |>(4)
    //              |       BLOCK          |
    //          (1)<|                      |>(5)
    //                  ------------------
    //                          \/(2)

    // x1, x2 - input at pin 0 and pin1
    // y1, y2 - output at pin 4 and pin5
    // b1 - FIR Coefficients of the filter at input pin 2,
    // a1 - IIR Coefficients of the filter at input pin 2
    // b2 - FIR Coefficients of the filter at output pin 3,
```

```
    // a2 - IIR Coefficients of the filter at output pin 3

    // Para1, Para2 and Para3 are the variables that could be used
    // in the code and can be controlled from the block Dialog.

    for( int i = 0 ; i < 256 ; i++)
    {
      y2[i] = 0;
      for(int j=0 ; j < b1.length ; j++)
      {
        y2[i] += x1[i]*b1[j];
      }

    }
    // Paste your code here
    // compile the file
    // upload the .class file
  }
}
```

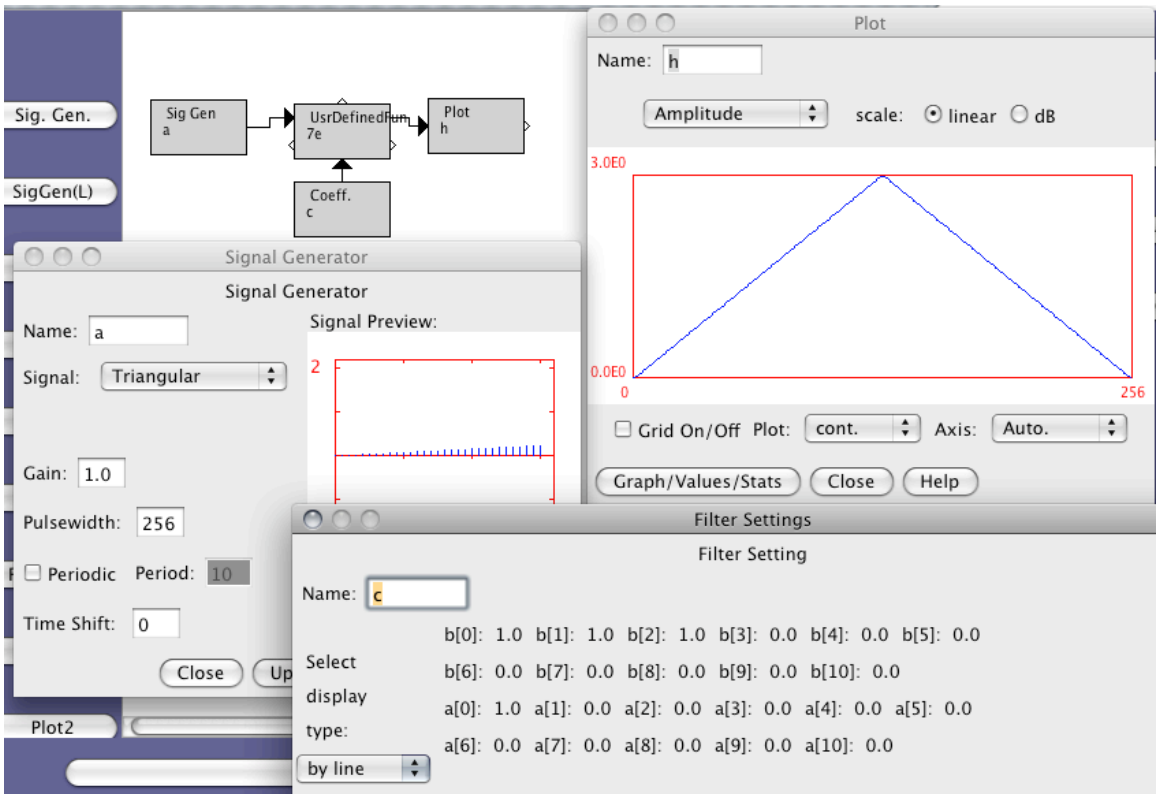


Figure 6.3: The result of using the weighted sum example code in J-DSP.

6.6 Bonus: More z-Transforms and Convolutions

The following properties of the z-transform may be useful here:

Property	Time Domain, $Z^{-1}\{\cdot\}$	z-Domain, $Z\{\cdot\}$
Linearity	$a_1x[n] + a_2y[n]$	$a_1X(z) + a_2Y(z)$
Time shift	$x[n-k]$	$z^{-k}X(z)$
Scaling in the z-domain	$a^n x[n]$	$X(a^{-1}z)$
Time reversal	$x[-n]$	$X(z^{-1})$
Differentiation in the z-domain	$nx[n]$	$-z\frac{\mathrm{d}X(z)}{\mathrm{d}z}$
Convolution	$x[n] * y[n]$	$X(z)Y(z)$

z-Transform of arbitrary sequences Give the z-transform of the following sequences, $x[n]$ (assume $x[n] = 0$ for all n not stated):

1. $x[n] = \{2, 4, 6, 4, 2\}$, $n = \{0, 1, 2, 3, 4\}$
2. $x[n] = \delta[n]$
3. $x[n] = \delta[n-1]$
4. $x[n] = 2\delta[n] - 3\delta[n-1] + 4\delta[n-3]$
5. $x[n] = 2\delta[n] + 4\delta[n-1] + 6\delta[n-2] + 4\delta[n-3] + 2\delta[n-4]$

Inverse z-transforms Give the inverse z-transform of $H(z) = 1 + 5z^{-1} - 3z^{-2} + 2.5z^{-3} + 4z^{-8}$.

Transfer function & impulse response Give the transfer function and impulse response for a filter with zeros at $(r, \hat{\omega}) = \{(1, 0), (1, \pm\pi/2), (0.9, \pm\pi/3)\}$.

7 Feedback Filters

By the end of this lab you should feel comfortable manipulating and using feedback filters for simple algorithms. You should also be comfortable with the concept of a filter with an Infinite Impulse Response (IIR). All feedback filters have an infinite impulse response and are also known as IIR filters. Feedback filters use the previous outputs of the filter, feeding them back into the output of the current sample. The "fed back" outputs are weighted by coefficients, a_ℓ.

7.1 Lab Background

Note that the J-DSP ***Coeff.*** block uses *negative* values for the a_ℓ (feedback) coefficients. In other words, in the text, a second-order feedback filter's defining equation might be:

$$y[n] = a_1 y[n-1] + a_2 y[n-2] + b_0 x[n] \tag{24}$$

$$y[n] - a_1 y[n-1] - a_2 y[n-2] = b_0 x[n] \tag{25}$$

This yields the transfer function:

$$Y(z)(1 - a_1 z^{-1} - a_2 z^{-2}) = b_0 X(z) \tag{26}$$

$$Y(z)/X(z) = \frac{b_0}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{27}$$

$$H(z) = \frac{b_0}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{28}$$

However, in J-DSP, the coefficients used, rather than being the a_ℓ from the defining equation (24) are the *negative* a_ℓ from the transfer function (28). In other words, to properly define the above filter in J-DSP, you will need to use the coefficients, $a[0] = 1, a[1] = -a_1, a[2] = -a_2$, and $b[0] = b_0$. Note also that in this example the ***Coeff.*** block has an $a[0]$ coefficient, which we will always leave as 1.0 (it's the first "1" in the denominator of the transfer function).

7.2 Feedback Filters as Recurrence Relations

You may notice that the defining equation for a feedback filter is in the form of a recurrence relation. In fact, we can use a feedback filter to implement a recurrence relation if we set the input to be an impulse, $x[n] = C\delta[n]$, with amplitude C being the initial value for the iteration. Let's start out with the Fibonacci sequence, which you'll remember to be:

$$F[n] = \begin{cases} 1 & n < 2 \\ F[n-1] + F[n-2] & n \geq 2 \end{cases} \tag{29}$$

We can rewrite this recurrence relation as:

$$y[n] = y[n-1] + y[n-2] + x[n] \tag{30}$$

and we will get the Fibonacci sequence *if* we input an impulse (hence, the appearance of the $x[n]$ on the right hand side, which serves only to initialize the filter). This is also a very good demonstration of the first "I" in the acronym "IIR": the impulse response of this filter has infinite duration.

Step 1.1 If we set $x[n] = \delta[n]$ in (30), we should see that the impulse response of this filter is indeed the Fibonacci sequence. Implement this filter in J-DSP and verify that its impulse response is the Fibonacci sequence. What are the values of the coefficients in the ***Coeff.*** block that you used?

Step 1.2 What is the value for $n = 19$ given by the ***Plot*** block?

Step 1.3 Is this filter stable?

Step 1.4 Let's do something similar with the recurrence relation for computing the series $y[n] = 1/3^n$ in the text. Set the coefficients for a feedback filter to implement equation (5-42) in the text, $y[n] = 1/3y[n-1] + x[n]$. What are the J-DSP filter coefficients?

Step 1.5 What are the pole location(s) for this filter?

Step 1.6 Now use J-DSP to calculate the impulse repsonse. Set the amplitude of the input impulse to be 0.99996. Is this filter stable? Is its impulse response consistent with the result of iterating equation (5-42) in the notes?

7.3 Telephone Touch Tone Dialing

Telephone touch pads generate dual tone multi frequency (DTMF) signals to dial a telephone. When any key is pressed, the tones of the corresponding column and row in the table below are generated, hence it is a "dual tone" code. As an example, pressing the 5 button generates the tones 770Hz and 1336Hz summed together.

	1209Hz	1336Hz	1477Hz
697Hz	1	2	3
770Hz	4	5	6
852Hz	7	8	9
941Hz	$*$	0	#

The frequencies in the table above were chosen to avoid harmonics. No frequency is a multiple of another, the difference between any two frequencies does not equal any of the frequencies, and the sum of any two frequencies does not equal any of the frequencies.[1] This

[1] More information can be found at: http://en.wikipedia.org/wiki/DTMF

makes it easier to detect exactly which tones are present in the dial signal in the presence of line distortions.

It is possible to decode by first using a *filter bank* composed of seven bandpass filters, one for each of the frequencies above. When a button is pressed, it will produce a combination of two tones, and thus, at the decoder end, two of the bandpass filters will produce higher outputs than the others. A good measure of the output levels is the average power at the filter outputs. This is calculated by squaring the filter outputs and averaging over a short time interval.

Step 2.1 J-DSP has a ***DTMF Tones*** block under the **Audio Effects** menu. You will use this as the input to your decoder. It generates DTMF tones, as you would expect, with a sampling rate of 8kHz. You'll probably want to set ***DTMF Tones*** to output five frames per button press, so that the tones are long enough to hear well. Convert the seven touch tone frequencies from Hz to digital frequencies in the range $[0, \pi]$.

Step 2.2 In this step, you will construct a bandpass filter for the 697Hz tone. Use a feedback filter with complex conjugate poles. Place a pair of complex conjugate poles using the ***PZ Placement*** block at the correct location for $\pm$697Hz (for manual entry, which is what you'll want to use, note that the phase angle is entered in degrees). Use equation (5-38) of section 5.1.4 of the text to set the radius of those poles so that the closest tone frequency, 770Hz, lies outside the passband (in other words, to set the bandwidth so that it is significantly smaller than twice the difference between 697Hz and 770Hz). You can verify this using the ***Freq-Resp*** block.

What were your pole locations?

Verify that the output for buttons [1], [2], and [3] are pretty much identical, and that all other buttons produce much lower amplitude output. In your report, include a plot of the filter output for one of the buttons [1], [2], or [3] and a plot for one of the buttons [4], [5], or [6].

Step 2.3 Now we are ready to decide whether a particular frequency is present. Run the filter output through a ***Square*** block (under the **Arithmetic** menu) and its output in turn through a ***Statistics*** block (under the **Basic Blocks** menu). Note that the ***Square*** block has two inputs; make sure you know which one you're using and that it has a "Coefficient" of 1.0. Examine the mean of the squared signal as you press the different ***DTMF Tones*** buttons. You should see a much higher mean squared value for buttons [1], [2], and [3] than the others. Additionally, the mean squared values for those three buttons should be almost identical. What are the mean squared values you get for pressing [1] versus [4]?

Step 2.4 Now we will assemble a filter bank. Your final layout should look something like Figure 7.1. Pare down your filter, etc. to the minimum set of blocks: ***Filter***, ***PZ Placement***, ***Square***, and ***Statistics***. Place four of these blocks down for the four frequencies 697Hz, 1209Hz, 1336Hz, and 1477Hz; this will allow us to recognize [1], [2], and [3]. Use

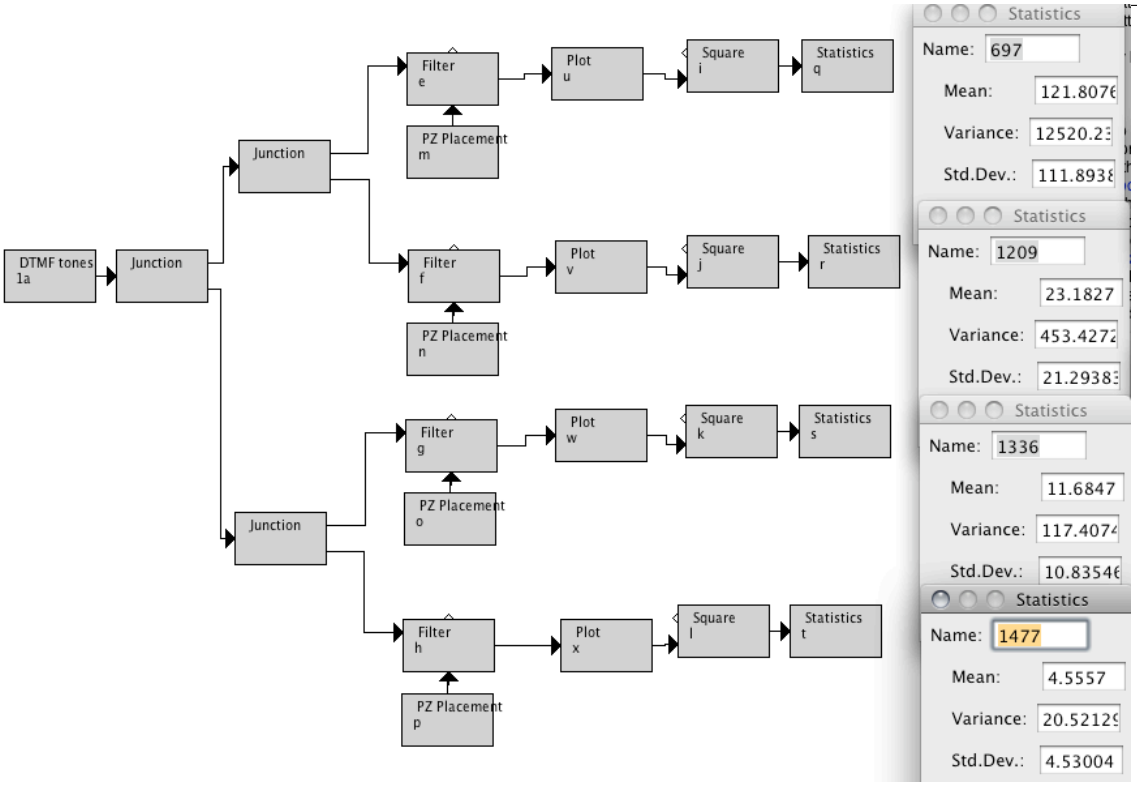


Figure 7.1: Example filter bank layout.

three ***Junction***s to split the output of the ***DTMF Tones*** block to send to four filters. Place the poles for each filter at the same radius and the appropriate angles for each frequency (remember, the ***PZ Placement*** block uses degrees for manual angle entry). Name the ***Statistics*** blocks for the filters' frequencies. Include a screen shot showing the output for ***DTMF Tones*** button [2] in your report. What are the mean squared values for each frequency for each of the ***DTMF Tones*** buttons [1], [2], [3], and [4]? Would you be able to use a simple pair of comparisons for each button to decode which button was pressed? Why or why not?

8 Discrete Fourier Transform

8.1 Lab Background

By the end of this lab you should have a firm understanding of how the Discrete Fourier Transform (DFT) can be implemented exactly using the Fast Fourier Transform (FFT). In addition you should be able to identify common problems using the DFT to analyze signals. You will also be familiar with a new tool, the spectrogram, that uses the DFT as a function of time.

8.2 Implementing the DFT

Recall that the DFT can be implemented directly from the analysis equation. For a length N signal $x[n]$,

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}nk} \qquad \text{for } k = 0, 1, 2, \cdots N-1 \tag{31}$$

The order of the implementation is $O(N) = N^2$. The following java code can be used to implement the DFT in J-DSP. It is implemented without any algorithmic speedup (i.e., it exactly mirrors equation 31). Example of raw DFT implementation:

```
public class MyFunction1
{
 public void myCode(double[] x1, double[] x2, double[] y1, double[] y2,
        double[] b1, double[] a1, double[] b2, double[] a2,
        double para1, double para2, double para3)
  {
  // x1, x2 - input at pin 0 and pin1
  // y1, y2 - output at pin 4 and pin5
  // raw DFT implementation
  double[] yImag;
  double[] yReal;
  yImag = new double[256];
  yReal = new double[256];
  double twoPiOverN = 2*Math.PI/256;
  for( int k = 0 ; k < 256 ; k++)
  {
    yReal[k] = 0;
    yImag[k] = 0;
    for( int n = 0 ; n < 256 ; n++)
    {
      yReal[k] += x1[n]*Math.cos(n*k*twoPiOverN);
      yImag[k] += -x1[n]*Math.sin(n*k*twoPiOverN);
    }
```

```
      y1[k] = Math.sqrt(yReal[k]*yReal[k]+yImag[k]*yImag[k]);
    }
  }
}
```

Note that there is not a native support in java for complex numbers so this arithmetic is written out explicitly in the code above. For example, the equation $y = x \times e^a$ (where x is a real number) must be explicitly written out using Euler's formula, and the real and imaginary portions saved in separate variables, $y_{real} = x \times \cos(a)$ and $y_{imag} = x \times \sin(a)$.

The FFT algorithm, on the other hand, can be used to reduce the computation time of the DFT to $O(N) = N \log_2 N$ - a significant speedup for even modest length signals.

Step 1.1 Write some example code to multiply *two* complex numbers, $c = a \times b$. Remember to represent the variables by their real and imaginary parts and save the real and imaginary parts in separate variables.

Step 1.2 Use the ***UserDefinedFun*** block to implement the FFT using loops (i.e., do not use recursion). The first part of this code performs a bit reversal on the input array. Use the following code to perform the bit reversal:

```
// bit reversal
int index;
int N = 256;
double[] reversedX1 = new double[256];
// need to shift 32 bit integer to make it an 8 bit reversal
int shift = (32-(int)Math.round(Math.log(N)/Math.log(2)));
for( int n = 0 ; n < N ; n++){
  index = (Integer.reverse(n))>>>shift; //reverse and then shift down to 8 bit
  reversedX1[n] = x1[index];
}
```

Then iterate over the array $\log_2 N$ times! Remember to explicitly carry out any complex arithmetic and take the magnitude of the output array once the FFT is computed. Include a copy of the java code in your report.

Step 1.3 Check your results in J-DSP using the ***FFT*** block. Your results should be identical. Take the FFT of a sinusoid with a frequency of $\pi/4$ radians per second using your FFT implementation and the ***FFT*** block provided in J-DSP. Include a screenshot of the output plots.

8.3 Using the DFT

Step 2.1 Create a sum of two sinusoids in J-DSP. Use the ***FFT*** block to compute the FFT of the sum and then plot the FFT magnitude. Use a frequency of 0.13π and 0.19π for

the two sinusoids. Make sure that the "pulsewidth" for each is set to 256. What does the result look like? Does this make sense?

Step 2.2 At what index (or indices) does the FFT magnitude reach its peak value(s)? What frequency (or frequencies) does this correspond to?

Step 2.3 Change the frequencies of the sinusoids to 0.13π and 0.14π. Repeat steps 2.1 and 2.2. Do the results still make sense?

Step 2.4 Replace the sum of two sinusoids in J-DSP with the ***DTMF*** block under the **Audio Effects** function list. Press some of the keys in the ***DTMF*** block. Does the FFT block show separate frequencies for each button?

8.4 Spectrograms

Comparing FFT graphs (as in the step 2.4) can be difficult to do. But what if we could analyze the frequency content of a signal as a function of time? That would make it easier to see differences in frequency if a signal started changing (like a string of DTMF keys pressed in turn). To do this we will need a new tool called the *Spectrogram*. The spectrogram is simply an algorithm for computing the FFT of a signal at different times and plotting them as a function of time. The spectrogram is computed in the following way:

1. A given signal is "windowed." This means that we only take a certain number of points from the signal (for this example assume we are using a window of length 128 points). To start out, we take the first 128 points of the signal (points 0 through 127 of the input array).

2. Take the FFT of the window and save the it in a separate array.

3. Advance the window in time by a certain number of points. For instance we can advance the window by 64 points so that we now have a window of indices 64 through 191 from the input signal array.

4. Repeat steps 1-3, saving the FFT of each window, until there are no longer any points in the input array.

5. Form a 2-D matrix whose columns are the FFT magnitudes of each window (placed in chronological order). In this way, each row represents a certain frequency, each column represents a given instant in time, and the value of the matrix represents the magnitude of the FFT.

The result is called a Spectrogram and is usually displayed as an RGB image where blue represents small FFT magnitudes and red represents larger FFT magnitudes (the *Jet* colormap if you are familiar with color visualizations). There is an art to choosing the correct

parameters of the spectrogram (i.e., window size, FFT size, how many points to advance the FFT, etc.). Each parameter has tradeoffs for the time and frequency resolution of the resulting spectrogram. For our purposes here, we will not be concerned with these tradeoffs. Instead we will be more interested in getting familiar with analysis using spectrograms.

Step 3.1 Use the DTMF setup from step 2.4. Instead of using the ***FFT*** block, use the ***Spectrogram*** block under the **Statistical DSP** function stack. Keep all parameters in the ***Spectrogram*** set to the default. Set the ***DTMF*** block to play five frames per key press. Press the number "1." What does the spectrogram look like? Are both frequencies present?

Step 3.2 Now set the ***DTMF*** block to "Record". Also set the "Resolution" parameter inside the ***Spectrogram*** block to 4 or 8 (otherwise you may notice a lag for updating the spectrogram - it can be considerably CPU intensive for large input signals). The "Resolution" parameter controls how many points the sliding window advances at each time step (larger steps mean we take fewer FFTs). Press each key in the ***DTMF*** block in turn. Does the spectrogram make it easier to judge the frequency content of the keys? Include a screenshot for your report.

9 Audio & Image Compression

9.1 Lab Background: sound and images in MATLAB

MATLAB has a number of functions that can be used to read and write sound and image files as well as manipulate and display them. See the `help` for each function for details. Ones that you'll likely find useful are:

audiorecorder Perform real-time audio capture. This may or may not work on your system; it is critically dependent on your hardware and MATLAB's support thereof. You may find it simpler to record and edit audio files with other software and then save it as `.wav` files to load into MATLAB (yes, that's right, MATLAB won't read `mp3` files).

sound Play vector as a sound. This allows you to create arbitrary waveforms mathematically (e.g., individual sinusoids, sums of sinusoids) and then play them through your speakers. This is the function to use if you're values are scaled within the range of -1 to +1.

soundsc This function works like `sound`, but first it scales the vector values to fall within the range of -1 to +1. Much of the time, you won't have your waveforms pre-scaled, and you'll use this function.

imread Read image from graphics file. MATLAB's image file support is much better than its audio file support. This function (and `imwrite`) supports many file types.

imwrite Write image to graphics file.

Within MATLAB, audio is a 1-D vector (stereo is $n \times 2$, but we won't deal with stereo) and grey-scale images are 2D arrays (color images are 3-D arrays). For simplicity's sake, make sure any sound files you use are mono.

9.1.1 Data Types

MATLAB supports a number of data types, and the type that the file I/O functions return often depends on the kind of file read. Regardless, almost all of the MATLAB functions you'll use to process data return doubles. For example, consider the following commands that read a color image, convert it to grey-scale, and display it:

```
>> A = imread('/tmp/ariel.jpg');
>> B = mean(A,3);
>> imagesc(B)
>> colormap(grey)
>> axis equal
>> whos
  Name      Size                          Bytes  Class
```

```
A       100x55x3                        16500  uint8 array
B       100x55                          44000  double array
```

The (color) image is read into `A`, which, as you can see from the output of the `whos` command, is a 100x55x3 unsigned, 8-bit integer array (the third dimension for the red, green, and blue image color components). I convert the image to grey-scale by taking the mean of the three colors at each pixel (the overall brightness), producing a `B` array that is `double`. The `imagesc` function displays the image, the `colormap` function determines how the array values translate into screen colors, and the `axis equal` command ensures that the pixels will be square on the screen.

Much of the time, we will just perform our calculations using `double` data types. However, we can use functions like `uint8` to convert our data. Note that the sound playing functions don't seem to like integer vectors.

At the very least, you can get real audio files from http://faculty.washington.edu/stiber/pubs/Signal-Computing/; I assume that you'll have no trouble locating interesting images (please keep your work G-rated). Don't use ones that are too big; there will be a 2MB size limit on E-Submit for your entire compressed file.

Write answers to any questions in the steps below in a file `answers.txt`, `answers.doc`, or something similarly named.

9.2 Lossless image coding

The simplest way to compress images is *run-length coding* (RLE), a form of repetitive sequence compression in which multiple pixels with the same values are converted into a count and a value. To implement this, we need to reserve a special image value — one that will never be used as a pixel value — as a flag to indicate that the next two numbers are a (count, value) pair, rather than just a couple pixels. Let's apply RLE to three different kinds of images: color photographs, color drawings, and black-and-white images (e.g., a scan of text). You can choose images you like, or get these from the book web site: http://faculty.washington.edu/stiber/pubs/Signal-Computing/ariel.jpg (color photo), http://faculty.washington.edu/stiber/pubs/Signal-Computing/cartoon.png (color drawing), and http://faculty.washington.edu/stiber/pubs/Signal-Computing/text.png (black-and-white).

Step 1.1 Write a MATLAB script to read an image in, convert it to grey-scale if the image array is 3-D, and scale the image values so they are in the range [1, 255] (note the absence of zero). Save your script as `step11.m`. Verify that this has worked by using the `min` and `max` functions. Convert the results to `uint8` and display each using `imagesc`. Save the resulting images as `step11a.jpg`, `step11b.jpg`, and `step11c.jpg`. If you used your own images, name them `step11a-in.jpg`, `step11b-in.jpg`, and `step11c-in.jpg`.

Step 1.2 At this point, you should have in each case a 2-D array with values in the range [1, 255] inclusive. To simplify matters, we will treat each array as though it were one-dimensional. This is easy in MATLAB, as we can index a 2-D array with a single index ranging from 1 to $N \times M$ (the array size). Write a RLE function (save it as `RLE.m`) that takes in a 2-D array and scans it for runs of pixels with the same value, producing a RLE vector on its output. When fewer than four pixels in a row have the same value, they should just appear in the vector. When four or more (up to 255) pixels in a row have the same value, they should be replaced with three vector elements: a zero (indicating that the next two elements are a run code, rather than ordinary pixel values), a count (should contain 4 to 255), and the pixel value for the run. Verify that your RLE function works by implementing a RLD function (RLE decoder, saved as `RLD.m`) that takes in a RLE vector, N, and M and outputs a 2-D $N \times M$ array. The RLD output should be identical to the RLE input (subtracting them should produce an array of zeros); verify that this is the case. For each image type, compute the compression factor by dividing the number of elements in the RLE vector by the number of elements in the original array. What compression factors do you get for each image?

9.3 Lossy audio coding: DPCM

In *differential pulse-code modulation* (DPCM), we encode the differences between signal samples in a limited number of bits. In this section, you'll take an audio signal, apply DPCM with differing numbers of bits, see how much space is saved, and hear if and how the sound is modified. A slight complicating factor is that all of the data will be represented using `double`, however, we will limit the values that are stored in each double to integers in the range $[0, 2^{\text{bits}} - 1]$ (where "bits" is the number of bits we're using).

Step 2.1 Find a sound to work with; you can use http://faculty.washington.edu/stiber/pubs/Signal-Computing/amoriole2.mat if you like. Likely, it will have values that are not integers; convert the values to 16-bit integer values by scaling (to $[0, 2^{16} - 1]$) and rounding (reminder: the vector's type will still be `double`; we're just changing the *values* in each element to be integers in that range). Verify that this conversion produces no audible change in the sound. What is the MATLAB code to do this initial quantization?

Step 2.2 Now write a DPCM function, saved as `DPCM.m`, that takes the sound vector and the number of bits for each difference and outputs a DPCM-coded vector. Note that the MATLAB `diff` function will compute the differences between elements of a vector. Your DPCM function should:

1. compute the differences between the samples,
2. limit each difference value to be in the range $[-2^{\text{bits}-1} - 1, 2^{\text{bits}-1} - 1]$, producing a quantized vector and a vector of "residues" (you will generate two vectors). For each quantized difference, the "residues" vector will be zero if the difference, Δx_i, is

within the above range. Otherwise, it will contain the amount that Δx_i exceeds that range (i.e., the difference between Δx_i and its quantized value, either $-2^{\text{bits}-1} - 1$ or $2^{\text{bits}-1} - 1$).

3. "make up for" each nonzero value in the "residues" vector. For each such value, scan the quantized difference vector from that index onward, and modify its entries, up to the quantization limits above, until all of the residue has been "used up".

4. The final result is a single coded vector that your function should return.

For example, let's say that we're using 4 bit DPCM and that some sequence of differences is $(\Delta x_1 = 10, \Delta x_2 = 5, \Delta x_3 = -3)$. The range of quantized differences is $[-7, +7]$, and so the quantized differences are $(7, 5, -3)$ and the residues are $(3, 0, 0)$. Since the first residue is nonzero, we proceed to modify quantized differences starting with the second one, until we've added three to them. The resulting final quantized differences are $(7, 7, -2)$.

Step 2.3 Implement an inverse DPCM function IDPCM, saved as `IDPCM.m`, that takes the first value from the original sound vector and the DPCM vector and returns a decoded sound vector. There's no way for us to know where the losses in the encoding occurred, so we just use the DPCM values as differences. Note that the MATLAB `cumsum` function computes a vector with elements being the cumulative sum of the elements of its input vector, and that you can add a constant to all of the elements of a vector using the normal addition operator.

Step 2.4 Test your DPCM and IDPCM functions by coding your sound using 15, 14, 12, and 10 bit differences. At what point does the coding process produce noticeable degradation (on cheap computer speakers)? Save your original sound vector as `step24-in.mat` and `step24-in.wav` and the IDPCM output sounds as `step24-15.mat` and `step24-15.wav`, `step24-14.mat` and `step24-14.wav`, `step24-12.mat` and `step24-12.wav`, and `step24-10.mat` and `step24-10.wav`. See the MATLAB `save` command for how to save individual variables in `.mat` files. For extra credit, investigate how few bits you can use to encode the sound and still detect some aspect of the original sound. Is this surprising to you?

9.4 Lossy image coding: JPEG

Step 3.1 In this sequence of steps, we will use frequency-dependent quantization, similar to that used in JPEG, to compress an image. Start with your grey-scale, continuous-tone image from step 1.1 (if you used a color image, convert it to grey-scale as you did in step 1.1). The MATLAB image processing or signal processing toolboxes are needed to have access to DCT functions, so we'll use the `fft2()` and `ifft2()` functions instead. To do a basic test of these functions, write a script that loads your image, converting it to grey-scale if necessary, and then computes its 2D FFT using `fft2()`. The resulting matrix has complex values, which we will need to preserve. Display the magnitude of the FFT (remember to use the `abs` function to get the magnitude of a complex number) using `imagesc()`. Do this for each image type. Can you relate any features in the FFT to characteristics in the original image?

Step 3.2 Use `ifft2()` to convert the FFT back and plot the result versus the original greyscale image (use `imagesc()`) to check that everything is working fine. Analyze the difference between the two images (i.e., actually subtract them) to satisfy yourself that any changes are merely small errors attributable to finite machine precision). Repeat this process for the other images.

Step 3.3 Let's quantize the image's spectral content. First, find the number of zero elements in the FFT, using something like `origZero=length(find(abs(a)==0));`, where `a` is the FFT. *Remember to exclude the DC value in the `fft2` output in figuring out this range.* Then, zero out additional frequency components by zeroing out all with magnitudes below some threshold. You'll want to set the threshold somewhere between the min and max magnitudes of `a`, which you can get as `mn=min(min(abs(a)));` and `mx=max(max(abs(a)));`. Let's make four tests, with thresholds 5%, 10%, 20%, and 50% of the way between the min and max, i.e., `th=0.05*(mx-mn)+mn;`. Zero out all FFT values below the threshold using something like:

```
b = a;
b(find(abs(a)<th)) = 0;
```

You can count the number of elements thresholded by finding the number of zero elements in `b` at this point and subtracting the number that were originally zero (i.e., `origZero`). This is an estimate of the amount the image could be compressed with an entropy coder. Express the number of thresholded elements as a fraction of the total number of pixels in the original image and make a table or plot of this value versus threshold level.

Extra credit. Note that the FFT may have very high values for just a few elements, and low values for others. You might plot a histogram to verify this. Not including the DC value likely will eliminate the highest value in the FFT. However, some of the other values may still be large enough to produce too large of a range. Can you suggest an approach that will take this into account? How does the JPEG algorithm deal with or avoid this problem?

Step 3.4 Now we will see the effect of this thresholding on image quality. Convert the thresholded FFT back to an image using something like `c = abs(ifft2(b));`. For each type of image and each threshold value, plot the original image and the final processed image. Compute the mean squared error (MSE) between the original and reconstructed image (mean squared error for matrices can be computed as `mean(mean((a-c).^2))`). What can you say about the effects on the image and MSE? Collect your code together as a script to automate the thresholding and reconstruction, so you can easily compute MSE for a number of thresholds. Plot MSE vs. threshold percentage (just as you plotted fraction of pixels thresholded vs. threshold in step 3.3).