Lab 3: Sounds in MATLAB

EE299 Winter 2008

Due: In lab, on or before 1 February 2008.

Objectives

In this lab we will cover:

- Representing, playing and plotting sound signals in MATLAB
- If statements and functions in MATLAB
- Altering/filtering sounds

Since sound signals are represented as vectors in MATLAB, you can do any mathematical operation on the sound signals that you could do on elements in a vector. In other words, you can create your own sounds with MATLAB scripts and functions. You will get to make a sound composition by modifying, mixing and stringing sounds together.

Prelab:

To ensure that you can complete everything within the allotted time, *before your lab section* you should:

- Read through the whole lab, and try to make some progress on exercises 1 and 2.
- Identify one or more sounds that you would like to use in your sound creations (exercises 3 and 4). (For sounds you create yourself, use m-files so that you don't have to recreate the idea from scratch when you find something you like.)

In order to finish in the allotted time, you should work with your teammates to combine the various sounds and modifications in exercises 3 and 4. You may create a single sound file as a team, or create individual ones that include shared component signals and/or processing techniques. You may want to plan out ahead of time who will do what.

1 Introduction to MATLAB (Part III)

1.1 Representing, Playing and Plotting Sampled Sound Signals in MATLAB

You have already been working with sounds in MATLAB, but mostly just following our examples. Here, we'll go over the MATLAB commands, in part as a review but also so that you'll understand the options you have.

Sounds can be on your computer in different formats. For example, .wav and .mp3 files are two particular formats for storing sounds, and sound-playing programs know how to read the files and produce sound using the computer's sound device. As we talked about in class, these formats all store a *sampled* signal, so the song is really just a long list of numbers, i.e. values of the signal at each sample time. In MATLAB, *mono* sounds can be represented as a really long vector, and *stereo* sounds as two really long vectors put together. There are a number of ways that we can get sounds into MATLAB, including:

- Convert an external sound file into a MATLAB vector, e.g. using the `wavread` command for sound files stored in the .wav format, as in either:
  ```
  >> [mySound Fs] = wavread('somesound.wav');
  >> [mySound Fs] = wavread('somesound');
  ```

- Load a sound signal that already exists as a MATLAB vector into your workspace, using the `load` command, as you did in Lab 1 with the "handel" clip. This is a way of re-loading "built-in" MATLAB variables, or variables that existed in a previous MATLAB session and that were saved in a file for future sessions. The syntax is either:
  ```
  >>load handel;
  >> load('handel');
  ```
  (The signal and sampling frequency are put into previously-defined variables, in this case `y` and `Fs`. The usual value of `Fs` for built-in MATLAB sounds is 8,192 Hz.)

- Create a vector from scratch in MATLAB. One way to do this is to use the function `makesinusoid.m`.

- Use the *wavrecord* function in MATLAB to record sound for the audio input of your sound card.

We will use all of these methods to generate vectors which store sounds. Once you have generated a vector, the original format doesn't matter, so plotting and playing sounds is the same for all cases. Format will matter again when you want to "write out" the sound (save it in a file for future use), as discussed at the end of this section.

As you did in previous labs, you can play a vector as a sound using the `sound` command. This command requires values to be in the range [-1, 1] or it will clip them to this range. (You can use `soundsc` instead to automatically scale values to this range. This avoids clipping, but changes the loudness of the sound.) If you want to play a sampled sound (in MATLAB or with other tools), you need to specify the playback rate (sampling rate) F_s in samples per second (Hz). (Recall $F_s = 1/T_s$, where T_s is the time between samples). In MATLAB, the function `sound` allows you to specify the sampling frequency as the second argument:

```
>> sound(mySound,Fs);
```

If you don't specify anything, it will use the default sampling frequency of 8192Hz (which was just fine for the Handel example). To convince yourself that this is important, try ignoring the sampling frequency for a CD-quality sound, as follows. Download the "Guitar and Castanets" sound file called "castanets44.wav" from the class web site under "Selected sound files." Convert it into a variable in MATLAB and play it by typing the following:

```
>> mySound = wavread('castanets44.wav');
>> sound(mySound);
```

You should notice that this sounds slower and lower pitch than what we've heard in class. This is because the playback rate is too slow for this sound. In order to play the sound back correctly, you need to explicitly read in the sampling rate. Now try:

```
>> [mySound,fs] = wavread('castanets44.wav');
>> sound(mySound);
>> sound(mySound,fs);
>> fs
```

The first version should sound the same as before, but the version with fs specified should sound like what you've heard before in class. By typing fs in MATLAB with no semicolon, you can see the actual sample rate. Is the result what you expect for CD quality? How different is it from the default?

An aside... Depending on the computer you are using, the sound card may have limitations on the sampling frequencies it can support. Many sound cards do not support sampling frequencies of less than 5-10kHz or greater than 44.1kHz.

You can plot the time signals using either `plot` or `stem`, where `plot` hides the discrete nature and `stem` makes it explicit. When you are plotting sampled signals as time functions, you should make sure that the appropriate time information is displayed on the time axis, for example using:

```
t=0:Ts:2;
```

```
plot(t,y)
```
where Ts=1/Fs. Another thing to keep in mind is that a very long signal will be hard to view in a single plot, especially a sinusoid with constant amplitude. It is often useful to plot only portions of a signal, or plot the signal in sections. You can display multiple plots at once using the function `subplot(n,m,k)`, which creates a $n \times m$ matrix of plot spaces in the figure and makes the k^{th} space ready for the next plot.

Once you have created a sound (or modified some sound that you read in), you can save it in the .wav format by using:

```
>> wavwrite(mySound,Fs,'filename');
```

where `mySound` is the vector your sound is stored in, `Fs` is the appropriate sample rate, and "filename.wav" is the name you want the file to have. If you have values outside [-1,1], they will be clipped and the function will return a warning message. If you don't specify the sampling frequency, it will assume a default of 8000 Hz. You can also specify the number of bits/sample, but we will just stick with the default (16 bits). If you want to write a stereo file, the format should be a matrix with 2 columns, one per channel.

1.2 *If* Statements in MATLAB

If is a programming tool that allows you to tell the computer to follow different instructions under different circumstances. If you already programmed in some other language you are likely familiar with it, but the syntax may be slightly different in MATLAB. The general form for a *if* statement is:

```
if expression that is either true or false
    . . . some MATLAB statements . . .
else
    . . . some MATLAB statements . . .
end
```

The first set of statements are done if the expression after `if` is true; otherwise the second set of statements (after `else`) are done. An extended form of the *if* statement that provides more than two options uses `elseif`. See `help if` to learn more about this.

If statements are included in several of the M-files that you will use in this lab. For example, in the M-file `makesinusoid.m`, it is used to test for problems with the input and respond differently to different problems. In the exercise below, you will use them to *clip* the sound signal (in other words, check that each value is within a desired range and replace it with a new value if it is not). Quantization of a signal can be implemented similarly.

In the exercise below we have provided the almost-complete script `filtersounds.m`, which uses 'for' loops and 'if' statements. If you aren't comfortable with the 'for' loops you should look back at Lab 2.

Exercise 1: Download the file `filtersounds.m` from the class webpage. This file contains the commands to do some simple modification and filtering of a sound signal, but it is not complete—you will need to complete it. The script should make three modified versions of the "handel" sound clip, by looping through all the samples in the sound vector and performing operations on them one at a time. In the low-passed version, each new sample becomes the average of the old values of that sample and its two nearest neighbors (one before, one after). Use the `mean` function to implement this. In the high-passed version, each new sample becomes the old value of that sample minus the values before and after. In the clipped version, the signal is clipped to the range -0.1 to 0.1, but values within that range are kept the same. The script plays the three filtered sounds. After it plays each one it pauses; you will need to hit `Enter` to continue. Show the TA your edited file and run it to play the sounds.

1.3 Functions in MATLAB

In Lab 2 you learned how to put a sequence of MATLAB commands into an M-file, which you can save or run later. Such a file is called a *script*. A *function* is also a sequence of commands saved in an M-file.

However, functions differ from scripts in that they have *input arguments* and *output values*. You can think of a function as a machine that takes a given input (or several inputs) and computes the output(s). You create a new function by writing the instructions in an M-file. The instructions say how to compute the output for any possible value of the input. The benefit of this is that you can run the function again and again with different input values, without having to edit the M-file.

MATLAB has many "built-in" functions that already exist, so you can use them without creating them. For example, in Labs 0 and 1, you used many built-in MATLAB functions like `size`, `length`, `ones`, `zeros`, `imagesc`, `sound`, `colormap`, `title`, etc. These functions were already defined in MATLAB: for example, the function `imagesc` takes a matrix as input and displays it as an image.

It is important to distinguish between creating (*defining*) the function and running it. When you write the instructions in an M-file you are defining the function. After defining it, you can run it by typing its name and giving values for the inputs, as described further below. For example, when you type `sound(y,Fs)` you are running the `sound` function with the particular values in your variables `y` and `Fs` as the inputs. Running a function is also referred to as *calling* it. Functions that already exist can be called from the Command Window, from scripts, or from other functions. As for script M-files, the function M-file does not have to be open for you to run it, but it must be "visible" to MATLAB; the easiest way to do this is to make sure the M-files are in the Current Directory.

Note again that the inputs are variables in the M-file and do not have pre-assigned values. In order to run the M-file, you need to call it with some input values. If you create a new function in an M-file and then click on the "run" icon at the top you will get an error, because MATLAB doesn't know what the values of the inputs are.

There is a standard way to define functions in MATLAB. The first line of the M-file is of the form:

`function [`*list of outputs*`]=`*functionname*`(`*list of inputs*`)`

For example, here is a very simple function, which would be saved in an M-file called `simpleFunction.m`:

```
function [x,y,z]=simpleFunction(number1,number2)
x=number1+number2;
y=number1*number2;
z=number2+10;
```

An example call to this function from the Command Window would be:

```
>>[out1,out2,out3]=simplefunction(3,4)
out1 =
  7
out2 =
  12
out3 =
  14
```

M-files should be saved as the same name as the function they contain, for example a M-file with containing the function named `function_name` should be saved as `function_name.m`.

When you call a function without the arguments it expects, or with arguments of different dimensions, MATLAB will give you an error message. However, many built-in MATLAB functions that implement mathematical functions can operate on either a scalar or a vector, producing either a scaler or vector output. For example, in MATLAB `cos(x)` is a built-in function whose input `x` is a number, and whose output is a number. However, you can give a vector input and get a vector output of the same size, where, say, element 3 in the output vector is the cosine of element 3 in the input vector. This is useful for generating signals.

Exercise 2: Download the function M-file `makesinusoid.m` from the EE299 web page. (Make sure it is saved as an M-file (".m") and that there is no additional text added by the browser.) Specify for the TA what the input variable(s) are and what the output variable(s) are. In the function, *if* statements are used to check for two possible problems, and the built-in MATLAB functions `disp` and `error` are used to give information about the problems to the user. Explain to the TA what these built-in functions do differently and why it makes sense to use each for displaying the different messages.

Note that M-files (functions or scripts) can call other M-files that you write, as well as the built in functions in MATLAB. You will explore this in the next exercise.

Exercise 3: Create a sound signal that is a sequence of multiple different signals, including at least one that you read in (from the class web page, or the built-in MATLAB sounds "chirp", "handel", "gong", "laughter", "splat", or "train") and at least one created from scratch. *Make sure that all the sounds you use have the same sampling frequency. You can change the sampling rate using the MATLAB* `resample` *command – use help to find out more if you need to use it.* For the one(s) you create, you can use the `makesinusoid` and `fade` functions and/or the script you created in Lab 2. You should use at least one function in your script. Concatenate the signals (using `sound=[X Y Z]` as in Lab 2), and then play the result for your TA. In concatenating, make sure that you pay attention to row vs. column format: `wavread` and `load` generate column vectors, `makesinusoid` generates a row vector, and `sound` requires a column vector. Use `subplot` to display, for your TA, sections of the different sounds included in the signal, and describe what you did for the signal you created.

2 Signal Processing Sound Effects

Signal processing provides many ways of modifying signals, which you may want to do to enhance the quality of the signal, transform it for communications or just to create sound effects or computer music. Some examples of simple modifications that we've discussed include:

- combining multiple signals by adding the vectors;
- concatenating multiple signals in sequence;
- introducing a delay (or, adding a silent region) by concatenating the signal with zeroes;
- changing the volume (amplitude scaling) by multiplying the signal by a scalar (the end result should be in the [-1,1] range if you play it with the `sound` command or write it out with `wavwrite`);
- creating an echo effect by repeating the signal (by concatenating amplitude-scaled copies of it, optionally with zeroes in between);
- multiplying with another sound signal (such as a cosine or a decaying ramp as in the `fade` function);
- time reversal (you can do this using the MATLAB `fliplr` function, which flips a row vector, or `flipud`, which flips a column vector);
- timescaling (e.g. throwing away samples to speed up a signal, or inserting samples to slow it down);
- clipping (as in Exercise 1); and
- filtering to remove frequency content (as in Exercise 1 or using the MATLAB `filter` command).

Many of these capabilities are implemented in the `soundmixer` tool, which is a MATLAB script available on the class web page. You will use it in the next exercise so that you can very easily explore different operations and get a sense for what they do. The tool also has a capability for generating some simple

sounds, including silence, cosines and noise, and it will allow you to record your own sounds to add to the mix. It also takes care of sampling rate differences between signals for you.

Exercise 4: Your job is to create a sound composition between 10 and 30 seconds long using the `soundmixer` tool and optionally your own scripts to implement modifications that it does not include (such as fading). First download any sounds that you might want to include in your composition, from the class webpage under "Selected sound files" or any others you find online. Then explore different types of modifications on different types of signals with the `soundmixer` tool. You should explore all of the options, but you need not use everything in your final composition. Do include multiple effects though. When you are done, save your song using the "save" button on the tool and play it for the TA. Write a short description of the different signal processing techniques that you used to create the final sound. **Submit the final audio file and description to the CollectIt drop box for Lab 3.**